TOWARDS TIGHTER INTEGRATION OF MACHINE LEARNING AND DISCRETE OPTIMIZATION

A Dissertation
Presented to
The Academic Faculty

By

Elias B. Khalil

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Computational Science & Engineering

Georgia Institute of Technology

May 2019

TOWARDS TIGHTER INTEGRATION OF MACHINE LEARNING AND DISCRETE OPTIMIZATION

Approved by:

Dr. Bistra Dilkina, Advisor
Department of Computer Science
University of Southern California

Dr. George L. Nemhauser
School of Industrial & Systems Engineering
Georgia Institute of Technology

Dr. Le Song
School of Computational Science & Engineering
Georgia Institute of Technology

Dr. Shabbir Ahmed
School of Industrial & Systems Engineering
Georgia Institute of Technology

Dr. Tuomas Sandholm
Computer Science Department
Carnegie Mellon University

Date Approved: March 11, 2019

L’homme n’est rien d’autre que ce qu’il se fait.

Jean-Paul Sartre

To my beloved parents.

ACKNOWLEDGEMENTS

My rather long journey at Georgia Tech would not have been nearly as enjoyable or fruitful without the support and camaraderie of many mentors and friends.

Bistra Dilkina has been a wonderful Ph.D. adviser to me. I still recall our first meeting in the CSE alcove (back when there were still couches), days after she'd joined GT in 2013. As an inexperienced master's student at the time, I had put together a mediocre proof of a certain proposition. Bistra immediately pointed out the flaw, which eventually led to a correct proof and my first paper. Since then, Bistra's rigor and attention to detail have been instrumental in shaping my work. Personally, Bistra has been my #1 advocate in the research community. This has helped me build strong connections with researchers across the world and will be crucial to my career going forward. Most importantly, Bistra was a thoughtful and considerate adviser, and for that, I will be most grateful.

George Nemhauser will always be a great inspiration to me. His energy and enthusiasm for research are contagious. I feel privileged to have worked closely with George over the years. I am also thankful to Le Song and Shabbir Ahmed for their continued involvement in my research. I appreciate Tuomas Sandholm's time and patience, as he has provided invaluable feedback on this dissertation and my proposal, as well as career advice. Polo Chau is my first mentor in research, taking me in when I had no experience at all; I will be forever grateful for his guidance and support, but most importantly for serving as an academic role model, inside and outside the classroom.

My two internships at IBM Research in Yorktown Heights, NY, were very fun and intellectually stimulating. Thank you Horst, Udayan and Deepak for being wonderful mentors and allowing me to explore new ideas, some of which have inspired the final chapter of this dissertation.

I had the fortune of sharing a house with roommates who have become close friends: thank you Fadi, Khaled, Bachir, Khoudor, Amandeep, Salim, Florian and Sebastian for

putting up with me! It is within CSE that I've met most of my Atlanta friends. Rakshit Trivedi is the best lab neighbor I could've asked for; I will always cherish our daily conversations on life, politics, science, and sports, often over coffee and during long nights trying to catch deadlines. My academic siblings, Amrita, Caleb, and Aaron were awesome "brothers in arms" in the eternal struggle to get our clusters and machines (barely) working. Lanssie Ma has been a great friend (and host, and student!); for better or worse, we have bonded through long discussions on GT procedures and graduation requirements, which we may have figured out by now. Despite his suspect taste in German football, Patrick Flick has been a superb friend, on and off the pitch. On that note, Friday Football outside Klaus has served as the best conclusion to hundreds of weeks at GT; thank you Santosh for establishing this tradition! The following, among others, have also helped make life and research much more pleasant: Lluis, Mehrdad, Bo, Ari, Ankit, Kyungjin, Robert, Shang, Fred, Eisha, Karl, Chirag, Georges, Sabra, Kasimir, Jiajia, Saurabh, Brian, Jordi, Hanjun, Yuyu, Shruti, Wenwen, Edward, Kimis, Tung, Samuel, Harsh, Srini.

The Lebanese and Arab friends I've made during grad school have made Atlanta a true "home away from home". Thank you Karen, Patrick, and Nour for the unending WhatsApp conversations and unparalleled puns. Fadi Jradi and Aly Megahed have been big brothers and mentors who were always ready to think through big decisions with me and share the snarkiest political memes. These people also strongly make the cut: Nancy, Caline, Abdallah & Laura, Joanna & Yann, Mosaab & Nisreen, Osman, Imad, Tony, Aida, Sarah, Elie, Sara, Yara, Jad, Emile, Nibel, Serge, Carmen, Assil, Lizzie, Hazar, Yousef, Yasser, Lubna, Iyed. Leya Hojeij remains one of the kindest people I've met; I will always privilege our friendship and deeply regret that it was cut short too soon.

Despite the distance, friends in Lebanon (and elsewhere) have remained close and warm to me, while constantly questioning the amount of time I had spent in grad school. I am particularly grateful to Charbel El-Helou, Michel Al-Haddad and Renée Gharios for being amazing lifetime friends.

Since 2012, I've been extremely fortunate to have Antoinette by my side. She is my best friend, confidante and lover. We have overcome great odds to be together, and I'm sure our future will be as bright as can be. I am grateful to my grandmother, cousins, uncles, and aunts for keeping me in their thoughts, despite the distance. My younger brother, Jad, is an inspiration to me; he has never shied away from a challenge and is bound to do great things. Most importantly, I will forever be in debt to my parents, Pierre and Marlène, for their unconditional love and support. They are my backbone, having taught me to be good to others, work hard and pursue my goals, whatever they may be. I hope you'll always be proud of me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

Discrete Optimization algorithms underlie intelligent decision-making in a wide variety of domains. From airline fleet scheduling to kidney exchanges and data center resource management, decisions are often modeled with binary on/off variables that are subject to operational and financial constraints. In fact, Nemhauser [100] estimated that at least half of the winners of the INFORMS Franz Edelman Prize [1] between 2000–2013 used discrete optimization in one form or the other, translating into billions of dollars in savings or profits.

This thesis introduces “Learning-Driven Algorithm Design", a novel paradigm for boosting the performance of discrete optimization algorithms by leveraging two types of data: the set of problem instances arising from the application of interest; and information generated while solving each instance. We develop Machine Learning (ML) approaches that have advanced the state-of-the-art in both exact integer programming solvers as well as heuristics.

First, we show how Mixed Integer Programming (MIP) solvers can benefit from tailored, efficient machine learning models, resulting in *data-driven* MIP branch-and-bound algorithms that are the first of their kind. This paradigm is developed for two fundamental algorithmic tasks in branch-and-bound: *branching* and *heuristic selection*. Our methods augment the solver with the ability to direct the search based on the characteristics of a particular problem instance and the state of the search, resulting in substantial speedups on a variety of problem classes, as well as common benchmarks.

Second, in the realm of heuristic algorithms, we design a deep reinforcement learning approach to the automated design of greedy algorithms for graph optimization problems with simple constraints. For more complex problems with general constraints, we design a novel recurrent neural network model that takes a set of integer programming instances and learns to turn fractional solutions into integer (feasible) solutions. In both settings, we show

[1] https://www.informs.org/Recognizing-Excellence/INFORMS-Prizes/Franz-Edelman-Award

that highly effective algorithms can be learned from a set of training instances, generalizing to unseen instances for various families of coverage, routing, assignment and satisfiability problems.

Third and last, we illustrate the potential for discrete optimization in machine learning. In particular, we study an adversarial machine learning problem in the context of binarized neural networks, a popular class of lightweight models with inherently discrete structure. We design an efficient combinatorial heuristic for perturbing inputs to a trained binarized neural network, such that the network predicts the wrong output. Our method generally outperforms the widely-used gradient-based heuristics on image classification datasets.

CHAPTER 1
INTRODUCTION

Automated decision-making is one of the pillars of Artificial Intelligence (AI). The discrete optimization solvers of today are powerful tools that can prescribe near-optimal decisions to highly complex problems that span domains as diverse as aircraft routing [18], wildlife conservation [38], sports scheduling [101], dose distribution [87] and kidney exchange [1], to mention a few. As such, improving the performance of combinatorial solvers can have a dramatic impact across various domains. These solvers have evolved considerably over the past two decades, and are now capable of handling problems with many thousands of variables and constraints, owing in large part to algorithmic advances.

Simultaneously, new decision-making tasks have been identified in emerging domains. For instance, the advent of autonomous vehicles and the new field of “computational sustainability" have generated discrete optimization problems of high societal priority that must be addressed. These problems have peculiar combinatorial structure, one that solvers may not be well-equipped to handle. Another characteristic of such upsurging applications is that they produce optimization problems at a large velocity (e.g. hourly planning for a fleet of autonomous vehicles). Towards achieving the next giant leap in the performance of discrete optimization solvers, these challenges – new combinatorial structures and high problem velocity – must be met.

Despite their immense positive impact on discrete optimization solvers, the algorithmic advancements brought about by the Operations Research (OR) community have also exposed inherent limitations in how solvers are designed. Most crucially, the various modules of a solver typically implement hand-designed rules or heuristics that achieve certain tasks. Such rules are designed by expert solver developers through a process of careful *manual tuning* on a set of benchmark problems. This process is clearly limited by the particular

development setting and problem set used for tuning.

In summary, discrete optimization solvers suffer from two main limitations that make them ill-suited for new problems arising in emerging domains, such as those mentioned earlier. On the one hand, algorithmic decisions within the solver are often implemented as handcrafted rules, the design of which requires trial-and-error on a limited benchmark problem set; the set may not include instances that are representative of the user's application. On the other hand, both exact discrete solvers and heuristic algorithms solve each new problem instance *de novo*, even when they have already encountered many similar instances arising from the same application domain.

This thesis introduces "Data-Driven Algorithm Design", a novel paradigm for boosting the performance of discrete optimization algorithms by leveraging two types of data: the set of problem instances arising from the application of interest; and information generated while solving each instance. We develop Machine Learning (ML) approaches that have advanced the state-of-the-art in both exact integer programming solvers as well as heuristic algorithms.

We believe that this dissertation constitutes a major step towards establishing ML as a central component of the algorithm design process, one that complements human ingenuity rather than replace it.

Thesis Statement

Augmenting Discrete Optimization algorithms–both exact and heuristic–with Machine Learning models significantly speeds up the solution of hard combinatorial problems arising in a wide variety of applications.

The rest of this chapter discusses the wide impact that discrete optimization has had in various domains, gives a brief survey on machine learning as it applies to optimization, presents basic algorithmic ideas for solving discrete optimization problems and overviews the structure of this dissertation and its contributions to the field.

1.1 Preliminaries

Definition 1 (Mixed Integer Program (MIP)) *Given matrix $A \in \mathbb{R}^{m \times n}$, vectors $b \in \mathbb{R}^m$ and $c \in \mathbb{R}^n$, and a subset $I \subseteq \{1, ..., n\}$, the* mixed integer program *MIP = (A, b, c, I) is:*

$$z^* = \min\{c^T x | Ax \leqslant b, x \in \mathbb{R}^n, x_j \in \mathbb{Z}\ \forall j \in I\}$$

The vectors in the set $X_{MIP} = \{x \in \mathbb{R}^n | Ax \leqslant b, x_j \in \mathbb{Z}\ \forall j \in I\}$ are called *feasible solutions* of the MIP. A feasible solution $x^* \in X_{MIP}$ is called *optimal* if its objective value $c^T x^*$ is equal to z^*.

Definition 2 (LP relaxation of a MIP) *The* linear programming (LP) relaxation *of a MIP is:*

$$\check{z} = \min\{c^T x | Ax \leqslant b, x \in \mathbb{R}^n\}$$

When $X_{MIP} \neq \emptyset$, $\check{z}$ is a lower bound for z^*, i.e. $\check{z} \leqslant z^*$.

Branch-and-Bound [102] is the main exact approach for solving MIP problems. It keeps a list of search *nodes*, each with a corresponding LP problem, obtained by relaxing the integrality requirements on the variables in I that have not been fixed to an integer value at an ancestor node. Let $\mathcal{L}$ denote the list of *active* nodes (i.e. nodes that have not been pruned nor branched on). Let $\bar{z}$ denote an upper bound on the optimum value z^*; initially, the bound $\bar{z}$ is set to ∞ or derived using heuristics that find an initial feasible solution. A lower (or dual) bound is derived by solving the LP relaxation of a MIP.

Two of the main decisions to be made during the algorithm are *node selection* and *variable selection*. In the former, the goal is to select an active node N_i from $\mathcal{L}$. Following that, the LP relaxation at N_i is solved, and its solution vector (if one exists) is $\check{x}^i$ with value $\check{z}^i$. N_i is pruned if its LP is infeasible, or if $\check{z}^i \geqslant \bar{z}$. If the LP solution $\check{x}^i$ is integer-feasible, i.e. $\check{x}^i \in X_{MIP}$, and $\check{z}^i < \bar{z}$, then z^* is updated to $\check{z}^i$, and $\check{x}^i$ is the new incumbent; the node is also pruned, as no better feasible solution can exist in its subtree.

If the selected node is not pruned, then it must be expanded into two child nodes. This is done by *branching* on an integer variable that has a fractional value in $\check{x}^i$, i.e. a variable $j \in I$ for which $\check{x}^i_j \notin \mathbb{Z}$, where $\check{x}^i_j$ denotes the value of variable j in the LP solution $\check{x}^i$. The two child nodes N^-_j and N^+_j are the result of branching on j downwards ($x_j \leqslant \lfloor \check{x}^i_j \rfloor$ in all descendants N_k) and upwards ($x_j \geqslant \lceil \check{x}^i_j \rceil$ in all descendants N_k). Variable selection deals with the problem of selecting that variable j from a set of possible candidates. We assume that MIP problems we treat are feasible, as real models are almost always so.

1.2 Related Work

Machine learning has been used in a variety of contexts within discrete optimization, as observed in surveys on the topic such as [118, 21]. A large body of work on "parameter configuration" has provided early evidence of the potential of tuning solvers to families of instances; see [115] for an example in combinatorial auctions. In [62, 64, 127, 63], the problems of solver parameter configuration and runtime prediction are studied; features of a MIP instance are used in combination with features of parameter configurations to predict the value of a performance metric (e.g. runtime) and guide a black-box parameter search algorithm. Parts I and II of this dissertation present a different approach to tailoring solvers to instances. The literature on parameter configuration focuses on selecting a good version of an algorithm from a (very large) space of possible algorithms; this can be seen as a high-level, coarse-grained search. Our work aims at improving solvers and algorithms at a much more fine-grained level, modifying their behavior at the level of their algorithmic components rather than high-level parameters.

Examples of work in the same spirit include using deep learning to tune gradient descent [14], reinforcement learning for job-shop scheduling [128], classification for selecting an algorithm for QBF subproblems [114], or regression to learn good restart rules for local search algorithms [28], to name a few.

In the MIP literature, a number of tasks within Branch-and-Bound have successfully

leveraged ML, including: branching [12], node selection [56], decompositions [83] and formulation selection [27]. We refer to the recent survey by Bengio et al. [21] for a structured exposition of these and many other papers.

1.3 Contributions & Structure of this dissertation

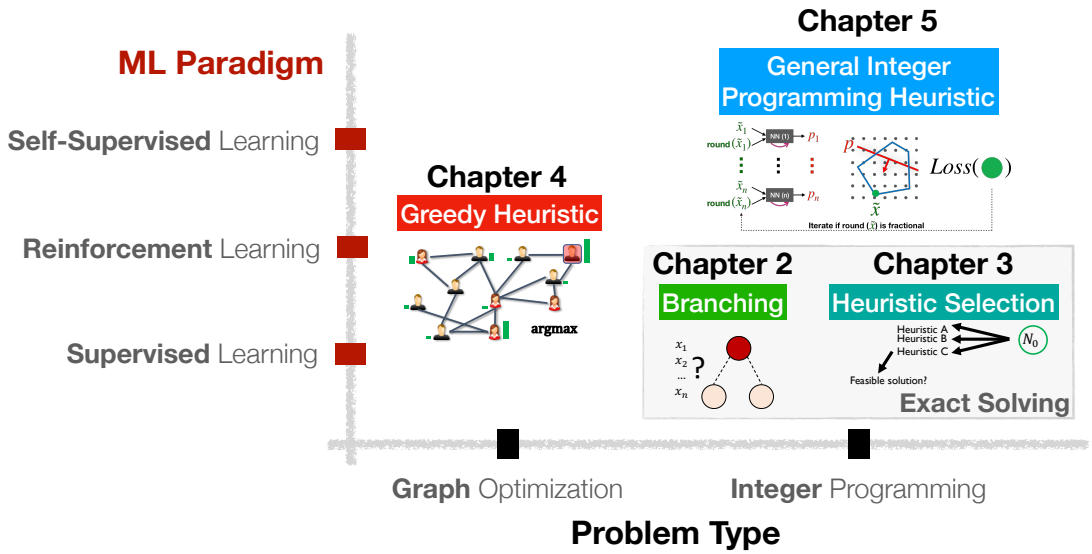


Figure 1.1: A thematic illustration of the chapters of Parts I and II.

In this section, we present the structure of this dissertation along with the contributions of each chapter. The dissertation is divided into three major parts.

Learning in Branch-and-Bound. Part I of this dissertation consists of Chapters 2 and 3, and examines the use of machine learning in exact branch-and-bound solvers for integer programming.

Chapter 2 studies “branching variable selection", a key component of MIP solvers. Choosing the right variables to branch on often leads to a dramatic reduction in the number of nodes needed to solve an instance. An ideal branching strategy (1) gives small search trees, and (2) is computationally efficient. We present the first effective machine learning approach to branching in MIP. Given an instance, a variable ranking model is learned

on-the-fly, during the early stages of search, and is applied thereafter to select a good variable. The learned model essentially approximates “strong branching", a very accurate but time-consuming ranking metric, with a cheap surrogate, thus simultaneously satisfying the desiderata above. Most notably, while existing strategies simply score variables based on static, fixed metrics, the learned branching strategy is *adaptive* to the structure of the instance. When used within CPLEX, a commercial MIP solver, the learned branching strategy dramatically reduces the search tree size compared to the widely-used pseudocost branching strategy on a heterogeneous benchmark set (MIPLIB2010). When applied to MIP instances from wildlife conservation planning and road infrastructure design, the learned branching strategy reduces the mean optimality gap from 12% and 15% to 1%, respectively, compared to pseudocost branching.

Chapter 3 studies the task of deciding which heuristics should be run during branch-and-bound. While proving optimality is a key trait of exact solvers, finding high-quality feasible solutions early in the search is at least as crucial. For that reason, MIP solvers use “primal heuristics" periodically during the search. However, the questions of *when* and *what* heuristics should be run during the search are handled heuristically via hard-coded rules: some heuristics are turned off by default, some run at every node and others every 10 nodes, etc. For instance, in SCIP, a state-of-the-art open-source MIP solver, some heuristics are turned off, others run frequently (e.g. at every node), while yet another subset runs occasionally (e.g. every 10 nodes). Such rigid rules are static, instance-oblivious, context-independent, and are unable to adapt to the state of the search. Additionally, the algorithmic differences between primal heuristics result in substantial variation in performance. For instance, diving and neighborhood search heuristics are much more computationally expensive than their rounding counterparts, but are generally more likely to find quality feasible solutions. Alternatively, a heuristic should be run when it is most likely to succeed, based on the problem instance’s characteristics and the state of the search. We study the problem of deciding at which node a heuristic should be run, such that the overall

(primal) performance of the solver is optimized. This is the first work that formalizes and systematically addresses this problem. We devise a theoretical framework for analyzing this decision-making question, proposed a machine learning approach for modeling heuristic success likelihood, and designed practical rules that leverage the ML models to dynamically decide whether to run a heuristic at each node of the search tree. This approach improves the primal performance of the SCIP solver by up to 6% on a set of heterogeneous benchmark instances. On instances of the “forest harvesting problem" from sustainability, the primal performance improves by up to 60%. Interestingly, these substantial improvements did not require designing new heuristics, but only intelligent, data-driven utilization of existing ones.

Contribution of Part I

- Supervised learning methods can effectively guide both the dual and primal sides of branch-and-bound: faster proof of optimality via better branching, and improved feasible solutions via better heuristic selection, respectively;
- A host of variable and node features are engineered to support the learning, resulting in simple yet powerful predictive models that accomplish the final tasks;
- The learning models are incorporated into state-of-the-art solvers, resulting in competitive results on benchmarks and specific families of instances;
- In some cases, the learned models exhibit interesting behavior that was not captured in the existing expert-designed rules.

Relevant Publications: Khalil, Le Bodic, Song, Nemhauser, and Dilkina [77], Khalil, Dilkina, Nemhauser, Ahmed, and Shao [75].

Learning Heuristics for Discrete Optimization. In Part II, we show how a heuristic can be tailored to a distribution of problem instances with Deep Learning. Heuristics are often the weapon of choice for practitioners when optimality guarantees are not required. The design of good heuristics for discrete optimization problems often requires significant specialized knowledge and trial-and-error. In many applications, the same optimization problem is solved repeatedly on a regular basis, maintaining the same problem structure but differing in

the data (e.g. constraint coefficients). This provides an opportunity for learning heuristic algorithms that exploit the structure of such recurring problems.

Chapter 4 proposes a unique combination of reinforcement learning and graph embedding that addresses this challenge. Rather than using a simple, static greedy rule to construct a solution to a graph optimization problem (e.g. a greedy insertion heuristic for the TSP), a *learned scoring function* is used instead. Reinforcement learning overcomes the difficulty of collecting training data (which requires solving an NP-Hard problem), while the graph embedding approach produces powerful node features that do not rely on feature engineering. The proposed framework can be applied to a diverse range of combinatorial optimization problems over graphs, such as the Minimum Vertex Cover, Maximum Cut and Traveling Salesman problems. The learned heuristics mostly dominate classical algorithms for these problems on a variety of graph distributions, and produce interesting solving behavior that is not typical of classical algorithms.

Chapter 5 goes beyond “simple" graph optimization problems. Despite the fact that Minimum Vertex Cover or the TSP are NP-Hard, they are simple enough in the sense that their constraints are not difficult to satisfy: a greedy construction heuristic can always find a feasible solution, as we show in Chapter 4. Unfortunately, many real-world problems exhibit more complex constraints that defy basic greedy heuristics. For instance, consider the “data center resource management" problem, in which we’d like to assign tasks to processors such that the processors’ memory, space or processing budgets are respected (while possible minimizing some cost function). Irrespective of the scoring function they use, greedy heuristics may overload a processor with tasks such that an unassigned task does not fit on any of the processors; to resolve this conflict, the algorithm must use some form of backtracking, which makes the learning of greedy algorithms, using the ideas developed in Chapter 4, very challenging. The key contribution of this work lies in incorporating projection (through linear programming) into a recurrent neural network model that generates solutions to a discrete optimization problem with arbitrary linear

constraints. Given a set of training instances from a problem of interest, such as the resource management problem discussed earlier, we tune the parameters of the recurrent neural network so that it produces projection directions that result in more integral solutions. This integrality criterion is captured by an appropriate loss function. Perhaps most importantly, the three components of our model– the recurrent network generating projection directions, the projection linear program (with appropriate smoothing) and the integrality loss function– are differentiable, allowing for training the model parameters through gradient descent. We refer to this heuristic-learning framework as “Neural Integer Optimization" (NIO). On assignment, knapsack and satisfiability type problems, the projection heuristics learned by NIO perform substantially better than the “Feasibility Pump", a widely used, non-learned heuristic for integer programming. Most notably, compared to a recently proposed neural network approach for satisfiability [116], NIO solves more (test) instances in fewer iterations after being trained on only hundreds of instances of a completely *different problem*; the approach in [116] requires *millions* of training instances. We view NIO as a promising approach to expanding the reach of machine learning to uncharted optimization problems where classical theoretical and algorithmic results are still underdeveloped.

Contribution of Part II

- Deep learning models can effectively guide discrete optimization heuristics;
- Supervised learning is often impossible when one must solve a problem instance to obtain a solution to learn from. Instead, reinforcement learning or self-supervised learning must be used;
- The learned heuristics often outperform hand-designed heuristics on a variety of combinatorial problems;
- In some cases, the learned heuristics exhibit novel solution-finding strategies that are not known in classical algorithm design, which could inform new approaches.

Relevant Publications: Dai, Khalil, Zhang, Dilkina, and Song [34] (equal contribution with Dai), Khalil, Trivedi, and Dilkina [78].

Discrete Optimization for Machine Learning. Parts I and II leverage machine learning in discrete optimization. But can discrete optimization benefit machine learning? Part III focuses on *discrete neural networks*, a popular class of ML models which requires rethinking continuous gradient-based optimization methods.

Recently, it has been shown that neural networks may be overly sensitive to tiny adversarial changes in the input, or “attacks". This weakness is detrimental to the use of these highly-accurate models in safety-critical domains. Designing attack algorithms that effectively fool trained models is a key step towards learning *robust neural networks*. I have designed a novel algorithm for attacking Binarized Neural Networks (BNNs), a class of *discrete deep neural networks* with binary parameters and threshold non-linearities. BNNs are popular due to their computational efficiency and potential for deployment to low-power devices. Attacking BNNs, and consequently protecting them, is thus an important problem in the emerging area of “adversarial machine learning".

The discrete, non-differentiable nature of BNNs, which distinguishes them from their full-precision counterparts, poses a challenge to standard gradient-based attacks. In Chapter 6, we study the problem of attacking a BNN through the lens of combinatorial and integer optimization [76]. An integer programming formulation of the problem is derived. While exact and flexible, the MIP quickly becomes intractable as the neural network grows larger. To address this issue, we designed a decomposition-based algorithm that solves a sequence of small MIP problems, thus scaling much better than the single global MIP. The proposed algorithm vastly outperforms the standard gradient-based attack on two image classification datasets, while simultaneously scaling far beyond a commercial solver on the global MIP.

Contribution of Part III

- Finding adversarial examples for Discrete (binarized) Neural Networks is challenging to existing gradient optimization methods;
- A novel decomposition-based combinatorial heuristic is developed, inspired by an exact MIP model;
- The proposed combinatorial methods show substantial improvement over gradient methods on image classification datasets.

Relevant Publication: Khalil, Gupta, and Dilkina [76].

Part I

Learning in Branch-and-Bound

CHAPTER 2
LEARNING TO BRANCH

The design of strategies for branching in Mixed Integer Programming (MIP) is guided by cycles of parameter tuning and offline experimentation on an extremely heterogeneous testbed, using the average performance. Once devised, these strategies (and their parameter settings) are essentially input-agnostic. To address these issues, we propose a machine learning (ML) framework for variable branching in MIP. Our method observes the decisions made by Strong Branching (SB), a time-consuming strategy that produces small search trees, collecting features that characterize the candidate branching variables at each node of the tree. Based on the collected data, we learn an easy-to-evaluate surrogate function that mimics the SB strategy, by means of solving a *learning-to-rank* problem, common in ML. The learned ranking function is then used for branching. The learning is instance-specific, and is performed on-the-fly while executing a branch-and-bound search to solve the instance. Experiments on benchmark instances indicate that our method produces significantly smaller search trees than existing heuristics, and is competitive with a state-of-the-art commercial solver.

2.1 Introduction

Variable selection for branching is considered to be a main component of modern MIP solvers [88, 7]. As part of the branch-and-bound algorithm for solving MIP problems [102], nodes in a search tree of partial assignments to variables must be expanded into (two) child nodes by selecting one of the unassigned variables and splitting its domain by adding additional constraints. Choosing good variables to branch on often leads to a dramatic reduction in terms of the number of nodes needed to solve an instance. In fact, a recent extensive computational study by researchers at IBM CPLEX, a leading commercial MIP

solver, shows that using a naive variable selection strategy degrades performance by a factor of more than 8, compared to modern strategies [7]. However, in the same study, the authors note that “the results show that some progress has been achieved in branching variable selection since CPLEX 8.0 (2002 version), but certainly no break-through" (p. 458).

Traditional branching strategies fall into two main classes: Strong Branching (SB) approaches exhaustively test variables at each node, and choose the best one with respect to closing the gap between the best bound and the current best feasible solution value. Achterberg [2] shows that SB can result in 65% fewer search tree nodes on average, compared to the state-of-the-art “hybrid branching" strategy. However, this comes at an increase of up to 44% in computation time, as more time is spent *per node*. On the other hand, Pseudocost (PC) branching strategies are engineered to imitate SB using a fraction of the computational effort, typically achieving a good trade-off between number of nodes and total time to solve a MIP. The design of such PC-based strategies has mostly been based on human intuition and extensive engineering, requiring significant manual tuning (initialization, statistical tests, tie-breaking, etc.). While that approach is important and constructive, we depart from it and propose to *learn* branching strategies directly from data.

We develop a novel framework for data-driven, on-the-fly design of variable selection strategies. By leveraging research in supervised ranking, we aim to produce strategies that gather the best of all properties: 1) using a small number of search nodes, approaching the good performance of SB, 2) maintaining a low computation footprint as in PC, and 3) selecting variables adaptively based on the properties of the given instance. In the context of a single branch-and-bound search, in a first phase, we observe the decisions made by SB, and collect: features that characterize variables at each node of the tree, and labels that discriminate among candidate branching variables. In a second phase, we learn an easy-to-evaluate surrogate function that mimics SB, by solving a *learning-to-rank* problem common in ML [89], with the collected data being used for training. In a third phase, the learned ranking function is used for branching. This supervised learning framework for

branching variable selection is illustrated in Figure 2.1.

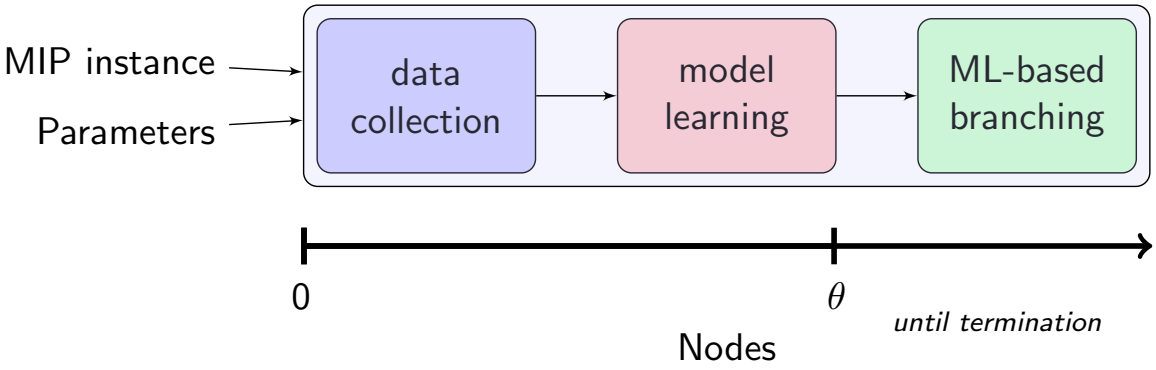


Figure 2.1: Proposed framework for learning to rank for variable selection: the axis represents the number of nodes processed during search, and θ is the number of nodes used for data collection and model learning.

Compared to recent machine learning methods for node and variable selection in MIP [56, 12], our approach: 1) can be applied to instances *on-the-fly*, without an upfront offline training phase on a large set of instances, and 2) consists of solving a ranking problem, as opposed to regression or classification, which are less appropriate for variable selection. Its on-the-fly nature has the benefit of being instance-specific and of continuing the branch-and-bound seamlessly, without losing work when switching between learning and prediction. The ranking formulation is natural for variable selection, since the reference strategy (SB) effectively ranks variables at a node by a score, and picks the top-ranked variable, i.e. the score itself is not important.

We will show an instantiation of this framework using CPLEX, a state-of-the-art commercial MIP solver. We use a set of static and dynamic features computed for each candidate variable at a node, and SVM-based learning to estimate a two-level ranking of good and bad variables based on SB scores. Experiments on benchmark instances indicate that our method produces significantly smaller search trees than PC-based heuristics, and is competitive with CPLEX's default strategy in terms of number of nodes.

2.2 The Branching Problem

A generic variable selection strategy can be described as follows. Given a node N_i whose LP solution $\check{x}^i$ is not integer-feasible, let $\mathcal{C}_i \subseteq \{j \in I | \check{x}^i_j \notin \mathbb{Z}\}$ be the set of branching candidates. For all candidates $j \in \mathcal{C}_i$, calculate a score $s_j \in \mathbb{R}$, and return an index $j^* \in \mathcal{C}_i$ with $s_{j^*} = \max_{j \in \mathcal{C}_i} s_j$. Two standard approaches to computing the variable scores are briefly described next; we refer to [2] for more details.

2.2.1 Strong Branching (SB)

Typically, the measure for the quality of branching on a variable x_j is the improvement in the dual bound. Consider a node N with LP value $\check{z}$, LP solution $\check{x}$, and candidate variable set $\mathcal{C}$. The two children N^-_j and N^+_j, resulting from branching on j downwards and upwards, have (feasible) LP values $\check{z}^-_j$ and $\check{z}^+_j$, respectively. If N^-_j (N^+_j) is infeasible, $\check{z}^-_j$ ($\check{z}^+_j$) is set to a very large value. The changes in objective value are then $\Delta^-_j = \check{z}^-_j - \check{z}$ and $\Delta^+_j = \check{z}^+_j - \check{z}$. To map these two values to a single score, let ϵ be a small constant (e.g. 10^{-6}), then:

$$SB_j = \text{score}\Big(\max\{\Delta^-_j, \epsilon\}, \max\{\Delta^+_j, \epsilon\} \Big) \tag{2.1}$$

A product is typically used for scoring, i.e. $\text{score}(a, b) = a \times b$. SB attempts to find the variable with the maximum score (2.1), by simulating the branching process for the candidate variables in $\mathcal{C}$, and computing the scores as in (2.1).

While SB directly optimizes (2.1), it is computationally expensive: solving two LP problems for each candidate variable using the simplex algorithm is often time-consuming. The time spent per node ends up overshadowing the time saved due to a smaller search tree. Simpler but faster heuristics are hence preferred.

2.2.2 Pseudocost Branching (PC)

Pseudocosts are historical quantities aggregated for each variable during the search. The upwards (downwards) PC of a variable x_j is the average unit objective gain taken over upwards (downwards) branchings on x_j in previous nodes; we refer to this quantity as Ψ_j^+ (Ψ_j^-). Pseudocost branching at node N with LP solution $\check{x}$ consists in computing values:

$$PC_j = \text{score}\Big((\check{x}_j - \lfloor \check{x}_j \rfloor)\Psi_j^-, (\lceil \check{x}_j \rceil - \check{x}_j)\Psi_j^+\Big) \tag{2.2}$$

and choosing the variable with the largest such value. As in SB, the product is used to combine the downwards and upwards values. One standard way to initialize the pseudocost values is by applying strong branching once for each integer variable, at the first node at which it is fractional [88]. We will refer to this PC strategy with SB initialization as *pseudocost branching* (PC).

As mentioned earlier, improving the performance of PC-based branching strategies requires significant offline tuning and experimentation on many instances. *Reliability branching* [6] is parametrized by a *reliability* threshold and a *lookahead* value. *Hybrid branching* [4] augments reliability branching with a tie-breaking mechanism, by combining the PC score of a variable with other scores; the different scores are scaled and weighted heuristically. Most recently, *hypothesis-reliability branching* [59] was proposed as an alternative to reliability branching, employing variance as a statistical measure of uncertainty in order to adaptively set the reliability parameter. Overall, PC-based strategies are input-agnostic, since the rule for selecting the variable to branch on is always the same (branch on the variable with largest PC score (2.2)), and is dependent on extensive parameter tuning on instances that may be arbitrarily different in structure from the input. Additionally, experiments show that PC-based strategies are still far from matching the node-efficiency of SB, requiring 65 to 75% more nodes than the latter, on average (Table 5.1, page 69 in [2]).

2.3 Overview of our Framework

We now introduce a framework for learning to branch in MIP. Our intuition is that by observing and recording the rankings induced by SB, we can learn a function of the variables' features that will rank them in a similar way, without the need for the expensive SB computations. Figure 2.1 illustrates our approach. Given a MIP instance and some parameters, we proceed in three phases:

1. Data collection: for a limited number of nodes θ, SB is used as a branching strategy. At each node, the computed SB scores are used to assign labels to the candidate variables; and corresponding variable features are also extracted. All information is compiled in a *training dataset*.

2. Model learning: the dataset is fed into a learning-to-rank algorithm that outputs a vector of weights for the features, such that some loss function is minimized over the training dataset.

3. ML-based branching: SB is no longer used, and the learned weight vector is used to score variables, branching on the one with maximum score until termination.

We highlight how the proposed method satisfies three desirable properties, before confirming so experimentally.

1. *Node-efficiency:* the learned ranking model uses SB scores and node-specific variable features as training data, and is thus expected to *imitate* the SB choices more closely than PC, yielding smaller search trees.

2. *Time-efficiency:* SB is used in the first phase only, typically for a few hundred nodes. The time required for learning (second phase) is small, and the third phase is dominated by feature computations, which are designed to be much cheaper than solving the LPs as does SB.

3. *Adaptiveness:* the learned ranking model is instance-specific, as it assigns different weights to features depending on the collected data.

Next, we describe each of the phases in more detail.

2.4 Data Collection

In this first phase, we aim to construct a training dataset from which we can learn a model that mimics SB's ranking. As such, the branch-and-bound algorithm is run with SB as the variable selection strategy, for a fixed number of nodes θ. If a node is fathomed (e.g. for infeasibility) during this phase, it does not count towards θ. At each node N_i, SB is run on a set of candidate variables $\mathcal{C}_i$, where $|\mathcal{C}_i| \leqslant \kappa$, and κ is typically in the range 10–20 in SB implementations (e.g. CPLEX). The variables in $\mathcal{C}_i$ are chosen among the fractional integer variables in the node's LP solution in standard ways (e.g. sorting by PC score [6]).

The training data then comprises:

- a set of search tree nodes $\mathcal{N} = \{N_1, ..., N_\theta\}$;
- a set of candidate variables $\mathcal{C}_i$ for a given node $N_i \in \mathcal{N}$;
- labels $\mathbf{y}^i = \{y^i_j \in \Omega | j \in \mathcal{C}_i\}$ for the candidate variables at each node i, where Ω is the domain of the labels;
- a feature map $\Phi : \mathcal{X} \times \mathcal{N} \to [0, 1]^p$, where $\mathcal{X} = \{x_1, ..., x_n\}$. $\Phi(x_j, N_i)$ describes variable x_j at node N_i with p features.

Notice how the same variable x_j may appear in both $\mathcal{C}_i$ and $\mathcal{C}_k$ for $i \neq k$, yet with different labels and feature values. This is a result of the choice of feature map Φ, which maps a variable *at the node in question* to features, capturing different contexts encountered during the search.

The specification and representation of the labels and features is a core issue when modeling a problem using machine learning. We present intuitive, simple guidelines for doing so in the context of branching.

Labels

A label is a value assigned to each variable in $\mathcal{C}_i$, such that better variables w.r.t. to the SB score have larger labels. We consider the SB score to be a sort of "gold standard" for scoring variables, hence labels based on such a score are a good target for learning.

We propose a simple and intuitive *binary labeling* scheme, i.e. $\Omega = \{0, 1\}$. Let $\mathbf{SB}^i$ denote the vector of SB scores for variables in $\mathcal{C}_i$ of node N_i, and $SB^i_* = \max_{j \in \mathcal{C}_i}\{SB^i_j\}$. A label y^i_j is computed by transforming the corresponding SB score SB^i_j as follows:

$$y^i_j = \begin{cases} 1, & \text{if } SB^i_j \geqslant (1-\alpha) \cdot SB^i_* \\ 0, & \text{otherwise} \end{cases} \tag{2.3}$$

where $\alpha \in [0, 1]$ is the fraction of the maximum SB score that a variable should have in order to get a '1' label. For instance, when $\alpha = 0.2$, variables whose SB scores are within 20% of the maximum score are assigned a label of '1'.

The labels resulting from the transformation in (2.3) emphasize our focus on the best variables w.r.t. SB, and are compatible with learning-to-rank formulations, as we will see in later subsections. While other labeling schemes are possible (e.g. grading on a scale of 1 to 5), we prefer the simple binary labels for the purposes of this work. Note that although our labels are 0/1, our setting is that of *bipartite ranking* (i.e. ranking with 0/1 labels), and not binary classification. Having a binary labeling scheme, as opposed to using a full ranking among all candidate variables, helps to avoid learning to correctly rank variables with low SB scores relative to each other, a task that is irrelevant to making a good branching choice. In addition, instead of assigning only the variable with maximum SB score a label of '1'

and all others a label of '0', our labeling scheme has a relaxed definition of "top" branching variable. This captures the fact that at some search nodes several candidate variables will have high SB scores, and will likely be good branching candidates.

Features

We compute a feature vector that describes a variable's state with respect to the node. The features can be split into two sets: *atomic features* are computed based on the node LP and candidate variable, whereas *interaction features* are the result of a product of two atomic features. The final feature vector can be seen as an explicit feature mapping, equivalent to the degree-2 polynomial kernel $K(y, z) = (y^T z + 1)^2$, in the space of atomic features.

The 72 atomic features are summarized in Table 3.2. They consist of counts and statistics (all or some of: mean, standard deviation (stdev.), minimum, maximum) capturing a variable's structural *role* within the node LP, as well as its historical performance. The atomic features are either *static* or *dynamic*. Static features are pre-computed once at the root node, and do not depend on the specific node LP.

For each feature, we normalize its values to the range $[0, 1]$ across the candidate variables at each node. This type of normalization is referred to as *query-based normalization* in the IR literature [90]. This produces an additional layer of dynamism, as the final value of a feature for a given variable depends on the set of candidate variables being considered at that node. For example, this results in static features of a variable taking on different normalized values across different nodes.

All atomic features can either be accessed through the solver API in $O(1)$, or computed in $O(nnz(A))$, i.e. in time linear in the number of non-zero elements of the coefficient matrix A, which makes data collection efficient.

Table 2.1: Description of the atomic features.

Feature	Description	Count	Reference
Static Features (18)			
Objective function coeffs.	Value of the coefficient (raw, positive only, negative only)	3	
Num. constraints	Number of constraints that the variable participates in (with a non-zero coefficient)	1	
Stats. for constraint degrees	The *degree of a constraint* is the number of variables that participate in it. A variable may participate in multiple constraints, and statistics over those constraints' degrees are used. The constraint degree is computed on the root LP (mean, stdev., min, max)	4	
Stats. for constraint coeffs.	A variable's positive (negative) coefficients in the constraints it participates in (count, mean, stdev., min, max)	10	
Dynamic Features (54)			
Slack and ceil distances	$\min\{\check{x}^i_j - \lfloor \check{x}^i_j \rfloor, \lceil \check{x}^i_j \rceil - \check{x}^i_j\}$ and $\lceil \check{x}^i_j \rceil - \check{x}^i_j$	2	
Pseudocosts	Upwards and downwards values, and their corresponding ratio, sum and product, weighted by the fractionality of x_j	5	[2]
Infeasibility statistics	Number and fraction of nodes for which applying SB to variable x_j led to one (two) infeasible children (during data collection)	4	
Stats. for constraint degrees	A dynamic variant of the static version above. Here, the constraint degrees are on the current node's LP. The ratios of the static mean, maximum and minimum to their dynamic counterparts are also features	7	
Min/max for ratios of constraint coeffs. to RHS	Minimum and maximum ratios across positive and negative right-hand-sides (RHS)	4	[12]
Min/max for one-to-all coefficient ratios	The statistics are over the ratios of a variable's coefficient, to the sum over all other variables' coefficients, for a given constraint. Four versions of these ratios are considered: positive (negative) coefficient to sum of positive (negative) coefficients	8	[12]
Stats. for active constraint coefficients	An active constraint at a node LP is one which is binding with equality at the optimum. We consider 4 weighting schemes for an active constraint: unit weight, inverse of the sum of the coefficients of all variables in constraint, inverse of the sum of the coefficients of only candidate variables in constraint, dual cost of the constraint. Given the absolute value of the coefficients of x_j in the active constraints, we compute the sum, mean, stdev., max. and min. of those values, for each of the weighting schemes. We also compute the weighted number of active constraints that x_j is in, with the same 4 weightings	24	[106]

2.5 Learning a Variable Ranking Function

Given the training data, we would like to learn a linear function of the features $f : \mathbb{R}^p \rightarrow \mathbb{R}$, $f(\Phi(x_j, N_i)) = \mathbf{w}^T \Phi(x_j, N_i)$ that minimizes a loss function over the training data. In ML terms, this problem is one of *empirical risk minimization*.

We define $\hat{\mathbf{y}}^i \in \mathbb{R}^\kappa$ to be the vector of values resulting from applying f to every variable in $\mathcal{C}_i$, i.e. $\hat{y}^i_j = f(\Phi(x_j, N_i))$. Formally, the learning problem can be written as that of finding:

$$\mathbf{w}^* = \underset{\mathbf{w} \in \mathbb{R}^p}{\arg\min} \sum_{N_i \in \mathcal{N}} \ell(\mathbf{y}^i, \hat{\mathbf{y}}^i) + \lambda \|\mathbf{w}\|_2^2 \tag{2.4}$$

The (structured) *loss function* $\ell : \mathbb{R}^\kappa \times \mathbb{R}^\kappa \rightarrow \mathbb{R}$ measures the loss resulting from ranking the variables at a node N_i according to $\hat{\mathbf{y}}^\mathbf{i}$, as opposed to the true labels $\mathbf{y}^i$, and $\lambda > 0$ is a regularization parameter that helps to avoid overfitting.

Fortunately, this problem has been studied in the context of web search, where a system is given a set of queries, and must rank a set of documents by order of relevance. We leverage existing research in information retrieval (IR) to address our problem (2.4). Specifically, our choice of the loss function $\ell(.)$ is based on *pairwise loss*, a common IR measure [67, 89].

A Pairwise Ranking Formulation

For a node N_i, consider the set of pairs: $\mathcal{P}_i = \left\{(x_j, x_k) \mid j, k \in \mathcal{C}_i \text{ and } y^i_j > y^i_k\right\}$. Each pair in any set $\mathcal{P}_i$ includes two variables at node N_i: one with label 1, and another with label 0. In order to rank variables similarly to how SB ranks them, we could learn an f that violates as few as possible of the following *pairwise ordering* constraints:

$$\forall i \in \{1, ..., \theta\} : \forall (x_j, x_k) \in \mathcal{P}_i : \hat{y}^i_j > \hat{y}^i_k \tag{2.5}$$

Violating a constraint in (2.5) is equivalent to learning a model for which $\hat{y}^i_j \leqslant \hat{y}^i_k$ for some node N_i and variables x_j, x_k: a variable with label $y^i_k = 0$ is ranked the same or higher than

one with label $y^i_j = 1$. Minimizing such violations means that the model is more likely to rank good variables higher than bad ones, which is our goal in variable selection. Note that other formulations for the *bipartite ranking* problem are also possible, e.g. [112], but the one we adopt is simple, can be optimized efficiently, and works well in practice.

While minimizing the number of violated constraints in (2.5) is NP-hard, Joachims [67] proposed a Support Vector Machine (SVM) approach that optimizes an upper bound on that number. SVM^{rank} is an efficient open-source package that implements that algorithm with the appropriate loss function ℓ [68]. We use SVM^{rank} with a cost-sensitive loss, weighting each term in the sum in (2.4) by $1/|\mathcal{P}_i|$ (parameter "-l 2"). This variant is more suitable, as it reduces the bias towards nodes with more "good" variables.

2.6 Branching with the Learned Function

After θ nodes have been processed, and the vector $\mathbf{w}$ has been learned, we switch from SB to the function f as variable selection strategy. At each new node N_i, we compute the feature vector $\Phi(x_j, N_i)$ for each variable $j \in \mathcal{C}_i$, and branch on the variable j^* with maximum score $s_{j^*} = \max_{j \in \mathcal{C}_i} s_j$, where $s_j = f(\Phi(x_j, N_i))$. The complexity of this procedure is $O(nnz(A) + p \cdot \kappa)$, where the first term is due to the computation of features, and the second is due to feature normalization and scoring by dot product (using p features and at most κ candidate variables per node). Compared to SB, which requires many dual simplex iterations, experiments show that our approach is much more efficient.

2.7 Experiments

2.7.1 Setup

We use the C API of IBM ILOG CPLEX 12.6.1 to implement various strategies using control callbacks, in single-thread mode. To evaluate the performance of any variable selection strategy $\mathcal{A}$, the strategy is run on a set of instances with a time cut-off of t_{max} seconds. An

instance $\mathcal{I}$ is *solved* by strategy $\mathcal{A}$ if and only if the run terminates within the tolerance gaps (we use default CPLEX values). If an instance $\mathcal{I}$ is not solved by the time cut-off, it is referred to as *unsolved*. All experiments were run on a cluster of four 64-core machines with AMD 2.4 GHz processors and 264 GB of memory; each run was limited to 2 GB of memory, and no run failed for memory reasons.

To isolate the effects of changing the variable selection strategy, we provide the optimal value as upper cutoff to CPLEX before the start of the search. This measure reduces the effect of node selection on the search, as the primal bound is given by the upper cutoff, and the order in which nodes are expanded has little impact on the tree itself. Additionally, cuts are allowed at the root only, and primal heuristics are disabled. These measures are common in branching studies [88, 45, 72], since they eliminate the interference between variable selection and other components of the solver, such as node selection. This also reduces *performance variability*, which we discuss in the next subsection.

Instances. We use the “Benchmark" set from MIPLIB2010 as our test set; we refer to [81] for details. This set was designed to span a variety of problem classes, applications, dimensions, levels of difficulty, etc., and is routinely used for evaluating branching strategies. The “Benchmark" set consists of 87 instances that can be solved by at least one commercial solver within 2 hours on a high-end PC. Note that since we turn off multi-threading and cuts beyond the root, we cannot expect to solve all instances within 2 hours. Hence, we set the time cut-off t_{max} to 5 hours (18,000 seconds). Three infeasible instances are excluded.

For each of the 84 instances we consider, we run every strategy with 10 different random seeds, for every variable selection strategy. Recent studies have shown that MIP solvers can be very sensitive to seemingly performance-neutral perturbations to their inputs [91, 7]. Therefore, runs with different seeds are necessary for obtaining meaningful results. In CPLEX, such perturbations can be induced by changing CPLEX’s internal random seed via its C API.

Branching strategies. We experiment with five strategies. CPLEX-D is the strategy that branches on the variable chosen by the solver with its default variable selection rule (as set by `CPX_PARAM_VARSEL`); this is done within a callback, as for all other strategies. Up until 2013, CPLEX developers report that the default selection rule is "a version of *hybrid branching*" [7]. SB refers to Strong Branching, while PC refers to pseudocost branching with SB initialization of the PC values [88]. SB+PC is a hybrid of SB for the first $\theta = 500$ nodes, and PC afterwards; a similar strategy appears in [45]. SB+ML is our proposed method with $\theta = 500$. We use $\alpha = 0.2$ and SVM^{rank} with a trade-off parameter $C = 0.1$ between training error and margin (λ in (2.4) is a function of C). We varied $\alpha \in \{0.1, 0.2, 0.3, 0.4, 0.5\}$ and $C \in \{0.001, 0.01, 0.1, 1\}$ (0.01 is the default in SVM^{rank}), and found that SB+ML performs similarly. For SB, SB+PC and SB+ML, κ is set to 10, and all SB calls are limited to 50 dual simplex iterations, as in [45].

We do not know what additional embellishments CPLEX uses, and report results for its default strategy as CPLEX-D. Even when CPLEX's default strategy is roughly known, callback implementations are much less node (and time) efficient; see Table 4 in [45] for an example. Hence, the main comparisons of our strategy are to PC and SB+PC. Most related to SB+ML is SB+PC, since both strategies share the same exact search tree up to θ nodes, then diverge by branching according to the variables selected by the learned ranking model and PC, respectively. Note that any extra information that CPLEX uses internally to score variables can be incorporated into our framework as features or labels, as can reliability branching scores, etc.

2.7.2 Results

We consider three metrics for evaluating branching strategies: the *number of unsolved instances*, the *number of nodes* to solve the instance and the *total time* to solve the instance.

Since our hypothesis is that our strategy SB+ML is better at variable selection, the main criterion for comparison is the number of nodes. The case for focusing on nodes versus time in benchmarking branching methods is discussed in depth in [61]. Total time is also important, and we have accordingly optimized our implementation of SB+ML to some extent. However, we believe that the time-efficiency of SB+ML may be improved, for example with access to CPLEX's internal data structures.

An instance with a different random seed is considered as a separate instance; this was suggested first by Danna [35]. Results for averages over seeds (per instance) are consistent with what we present here, but are not included due to space constraints. Of the 840 instances considered, we exclude: 184 instances solved by all strategies in 1,000 nodes (too easy), 82 instances not solved by any strategy in 5 hours (too hard), 3 instances that are flagged as infeasible by at least one strategy, and 48 instances for which CPLEX aborted. For Table 2.2, "All" refers to all instances considered, "Easy" and "Medium" refer to instances solved by CPLEX-D in less than 50,000 and 500,000 nodes, respectively; otherwise, an instance is classified as "Hard". When a strategy does not solve an instance, t_{max} is reported as the total time, and the number of nodes at termination is reported as the number of nodes. Note that this may be biased towards strategies that are slower, processing fewer nodes and solving fewer instances. This issue is inherent to MIP benchmarking [7], and we will address it in Tables 2.3 and 2.4. Similar experimental procedures are used in recent work on branching [45, 72].

Table 2.2 shows that SB+ML solves more instances than both PC and SB+PC. Most notably, SB+ML clearly outperforms PC and SB+PC, requiring respectively around 36% and 16% fewer nodes on average ("All" set). Currently, SB+ML spends, on average, 18 milliseconds (ms) per node, while SB, PC and SB+PC spend 68, 10 and 15 ms, respectively. Although SB+ML spends more time per node than SB+PC due to feature computations, it incurs a comparable total time, as it saves in the number of nodes. Compared to PC, our method is slower by 10% on average over all instances, but is 28% and 14% faster for

instances in “Medium" and “Hard", respectively. A more optimized implementation of the feature computations of SB+ML is likely to make it faster than competitors for the “Easy" set too. It is clear that SB is not applicable, as it requires twice as much total time as the three competing methods, and times out on many more instances.

Table 2.2: Summary of experimental results. “Unsolved instances" are counts, “Num. nodes" and “Total time" (in seconds) are shifted geometric means over instances with shifts 10 and 1, respectively. Lower is better, and the best value in each row among PC, SB+PC and SB+ML is in bold.

		CPLEX-D	SB	PC	SB+PC	SB+ML
Unsolved Instances	All (523)	11	129	66	63	**52**
	Easy (255)	0	12	15	14	**13**
	Medium (120)	2	43	22	22	**17**
	Hard (148)	9	74	29	27	**22**
Num. Nodes	All (523)	46,633	33,072	92,662	70,455	**59,223**
	Easy (255)	3,255	3,610	7,931	5,224	**5,124**
	Medium (120)	173,417	121,923	395,199	288,916	**234,093**
	Hard (148)	1,570,891	519,878	1,971,333	1,979,660	**1,314,263**
Total Time	All (523)	499	2,263	**960**	1,093	1,059
	Easy (255)	111	602	**243**	361	382
	Medium (120)	1,123	6,169	2,493	1,892	**1,776**
	Hard (148)	3,421	9,803	4,705	4,718	**4,039**

Table 2.3 addresses the bias in averaging over the number of nodes on unsolved instances in Table 2.2. Here, we consider *only instances solved by both strategies*, for every pair of strategies, and compute shifted geometric means on that subset of the instances. The node ratios shown in Table 2.3 are in line with the previous two tables. SB+ML needs 37% and 18% fewer nodes than PC and SB+PC, respectively, and only 3% more nodes than CPLEX-D. Interestingly, SB+ML requires 32% more nodes than SB, while CPLEX-D requires 39%; PC and SB+PC are dramatically worse.

Table 2.4 shows *win-tie-loss* counts for each pair of strategies, comparing the number of nodes needed to solve an instance head-to-head, and avoiding the averaging used in the two previous tables. An *absolute win* for strategy $\mathcal{A}$ over $\mathcal{B}$ on instance $\mathcal{I}$ is recorded iff

Table 2.3: Ratios for the shifted geometric means (shift 10) over nodes on instances solved by both strategies. The first value in a cell in row $\mathcal{A}$ and column $\mathcal{B}$ is the ratio of the average number of nodes used by $\mathcal{A}$ to that of $\mathcal{B}$. The second value is the number of instances solved by both $\mathcal{A}$ and $\mathcal{B}$.

	CPLEX-D	SB	PC	SB+PC	SB+ML
CPLEX-D		1.39 (389)	0.64 (449)	0.84 (452)	0.97 (463)
SB	0.72 (389)		0.47 (389)	0.61 (388)	0.76 (389)
PC	1.56 (449)	2.11 (389)		1.34 (445)	1.59 (450)
SB+PC	1.20 (452)	1.63 (388)	0.75 (445)		1.22 (454)
SB+ML	1.03 (463)	1.32 (389)	0.63 (450)	0.82 (454)	

Table 2.4: Win-tie-loss matrix for the number of nodes. A quintuple in a cell in row $\mathcal{A}$ and column $\mathcal{B}$ has: the number of absolute wins, wins, ties, losses and absolute losses for $\mathcal{A}$ against $\mathcal{B}$, w.r.t. the number of nodes.

	CPLEX-D	SB	PC	SB+PC
CPLEX-D				
SB	5/264/0/125/123			
PC	8/164/0/285/63	68/63/0/326/5		
SB+PC	8/227/0/225/60	72/66/7/315/6	15/320/0/125/12	
SB+ML	8/267/0/196/49	82/96/7/286/5	21/355/0/95/7	17/300/58/96/6

$\mathcal{A}$ solves $\mathcal{I}$ whereas $\mathcal{B}$ does not; a *win* occurs when both $\mathcal{A}$ and $\mathcal{B}$ solve $\mathcal{I}$, and $\mathcal{A}$ does so in strictly fewer nodes than $\mathcal{B}$; *absolute loss* and *loss* are defined analogously. A *tie* occurs when $\mathcal{A}$ and $\mathcal{B}$ solve $\mathcal{I}$ in the same number of nodes. Table 2.4 is consistent with Table 2.2, showing that SB+ML outperforms PC and SB+PC head-to-head, solving many more instances in fewer nodes. SB+ML has 21 absolute wins and 355 wins over PC, while PC has only 7 absolute wins and 95 wins over SB+ML in terms of number of nodes. SB+ML is on par with CPLEX-D in terms of overall wins, with fewer absolute wins (8 vs. 49), but more wins on instances solved by both (267 vs. 196).

Significance tests on aggregated seeds

Table 2.5: Win-loss counts for every pair of strategies, for the Wilcoxon signed rank test on the number of nodes. A doublet in a cell in row $\mathcal{A}$ and column $\mathcal{B}$ expresses the number of wins for $\mathcal{A}$ over $\mathcal{B}$, and the number of losses, respectively, w.r.t. to the outcome of the one-sided Wilcoxon test. $\mathcal{A}$ wins over $\mathcal{B}$ on instance $\mathcal{I}$ iff the null hypothesis H_0 (Strategy $\mathcal{A}$ solves $\mathcal{I}$ in more nodes than strategy $\mathcal{B}$) can be rejected at a significance level of 0.05; losses are defined analogously.

	CPLEX-D	PC	SB+PC
CPLEX-D			
PC	11/34		
SB+PC	19/26	32/6	
SB+ML	22/23	35/4	29/7

We evaluate whether a given branching strategy performs significantly better than another strategy in terms of the number of nodes on an instance, over multiple random seeds. This is different from the tables that appear in the main text, where runs with different seeds on the same instance are considered as instances of their own.

The one-sided Wilcoxon signed rank test is used here, as adapted to the MIP setting in [58, p. 40]. Let $\mathbf{n}^{\mathcal{A}}$ and $\mathbf{n}^{\mathcal{B}}$ denote the vectors containing the number of nodes for strategies $\mathcal{A}$ and $\mathcal{B}$ over a set of random seeds K on the same instance $\mathcal{I}$. The Wilcoxon test takes vectors $\log \mathbf{n}^{\mathcal{A}}$ and $\log \mathbf{n}^{\mathcal{B}}$ as input, and outputs a *p-value* corresponding to the probability of rejecting the following null hypothesis while it is true: the median of the distribution of the $\log\left(\mathbf{n}_k^{\mathcal{A}}/\mathbf{n}_k^{\mathcal{B}}\right)$ values (for all seeds $k \in K$) is negative. The Wilcoxon test uses the *ranks* of the entries of the $\log\left(\mathbf{n}_k^{\mathcal{A}}/\mathbf{n}_k^{\mathcal{B}}\right)$ vector to evaluate the null hypothesis. When $\mathcal{A}$ fails to solve $\mathcal{I}$ with seed k, the corresponding $\mathbf{n}_k^{\mathcal{A}}$ is set to a large value, such that $\mathcal{A}$ is penalized for this failure in the statistical test. The penalization is such that those failures are given the highest ranks, which is the desired outcome.

Table 2.5 shows that our method's improvement on the number of nodes is statistically significant compared to PC and SB+PC on many instances (35 and 29 instances, respectively), while PC and SB+PC beat SB+ML only on 4 and 7 instances, respectively. Compared to CPLEX-D, SB+ML wins 22 times versus CPLEX-D's 23, a much smaller difference compared to PC v/s CPLEX-D (11/34), and SB+ML v/s CPLEX-D (19/26).

Table 2.6: Win-tie-loss counts for every pair of strategies, for the number of nodes. A triplet in a cell in row $\mathcal{A}$ and column $\mathcal{B}$ expresses the number of wins for $\mathcal{A}$ over $\mathcal{B}$, the number of ties, and the number of losses of $\mathcal{A}$ to $\mathcal{B}$, respectively, w.r.t. to the shifted geometric mean of the number of nodes. $\mathcal{A}$ wins over $\mathcal{B}$ on instance $\mathcal{I}$ iff the mean number of nodes of $\mathcal{A}$ over the random seeds on $\mathcal{I}$ is strictly less than that of $\mathcal{B}$; losses and ties are defined analogously.

	CPLEX-D	PC	SB+PC
CPLEX-D			
PC	21/0/46		
SB+PC	28/0/39	46/0/21	
SB+ML	34/0/33	53/0/14	46/5/16

Table 2.6 shows that on more instances than not, SB+ML requires fewer nodes on average than CPLEX-D, PC and SB+PC. The improvement is most striking compared to PC and SB+PC.

Performance profiles

Performance profiles are commonly used in benchmarking optimization software and algorithms [40]. The profiles we present next are based on the same instances considered in the main text, i.e. without aggregation over seeds. We present a profile for the number of nodes in Figure 2.2. For a given strategy, a point (x, y) in Figure 2.2 is the fraction y of instances solved by that strategy within a factor of x times more nodes than the best strategy, for each instance. If a strategy fails to solve an instance, the ratio is set to a large value (twice the maximum ratio in the data, as per Dolan and Moré's code). The behavior of SB+ML in terms of nodes is almost indistinguishable from that of CPLEX-D up to a ratio (x-axis) of around 1.5. CPLEX-D then leads, while SB+ML dominates both SB+PC and PC. The profile for total time in Figure 2.3 is consistent with Table 2 in the main text, showing that SB+ML is less efficient than its competitors PC and SB+PC, a difference that we believe can be shrunk by optimizing the code.

2.7.3 Analysis of the Learned Models

Some interesting questions can be answered by analyzing the learned models: are the models "similar" across instances, suggesting that instance-specific learning is not useful? What

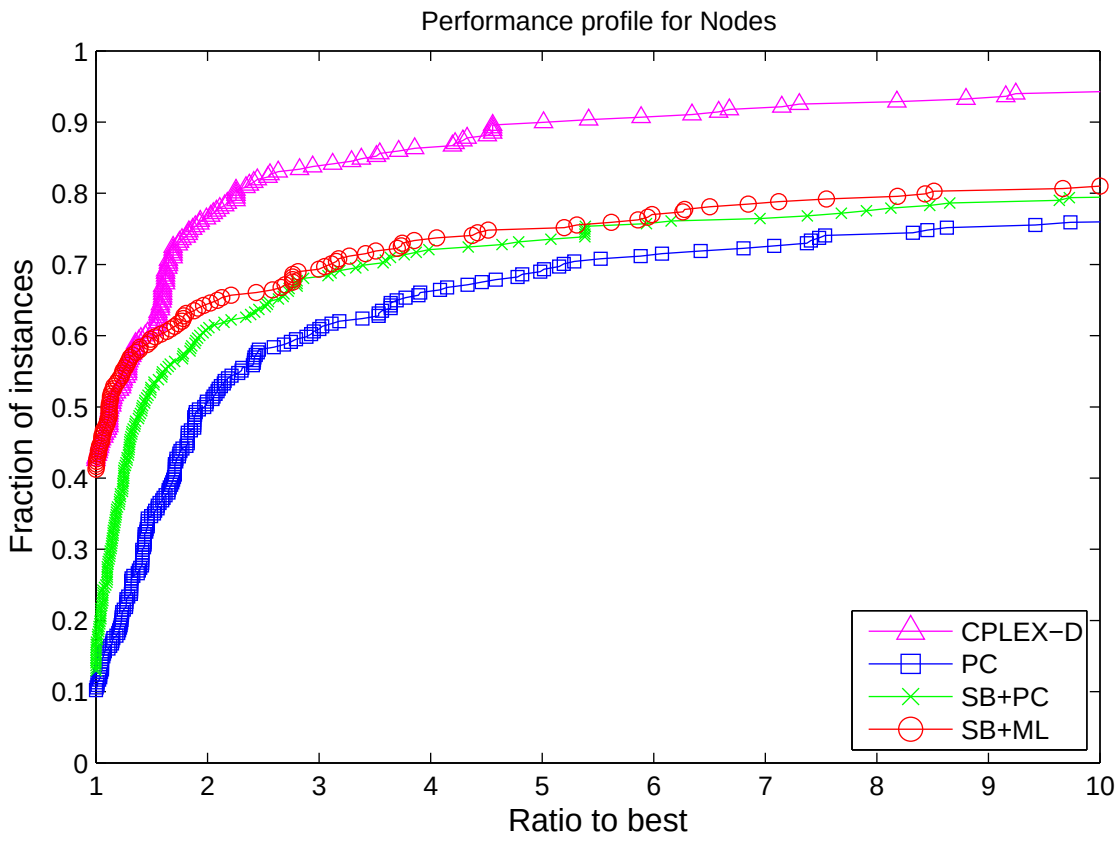


Figure 2.2: Performance profile for the number of nodes. For a given strategy, a point (x, y) on this graph is the fraction y of instances solved by that strategy within a factor of x times more nodes than the best strategy, for each instance.

features are the most predictive? To answer these questions, we perform an exploratory analysis on a subset of the learned ranking models. Specifically, we consider all learned models under one of the random seeds; there are 53 such models.

Similarity among models. Intuitively, two models are similar if ranking the features by their weights in each model produces two similar rankings; the actual weight values do not matter as much. First, we consider the 1378 unique pairs of models, for those 53 models that were learned. For each such pair (i, j), we compute Spearman's rank correlation coefficient [119], $\rho_{i,j}$, where $-1 \leqslant \rho_{i,j} \leqslant 1$. High positive values of $\rho_{i,j}$ indicate that the two models are highly correlated, i.e. the rankings of the features by their weights are similar, and vice versa for low negative values. Values of $\rho_{i,j}$ around zero indicate that the rankings

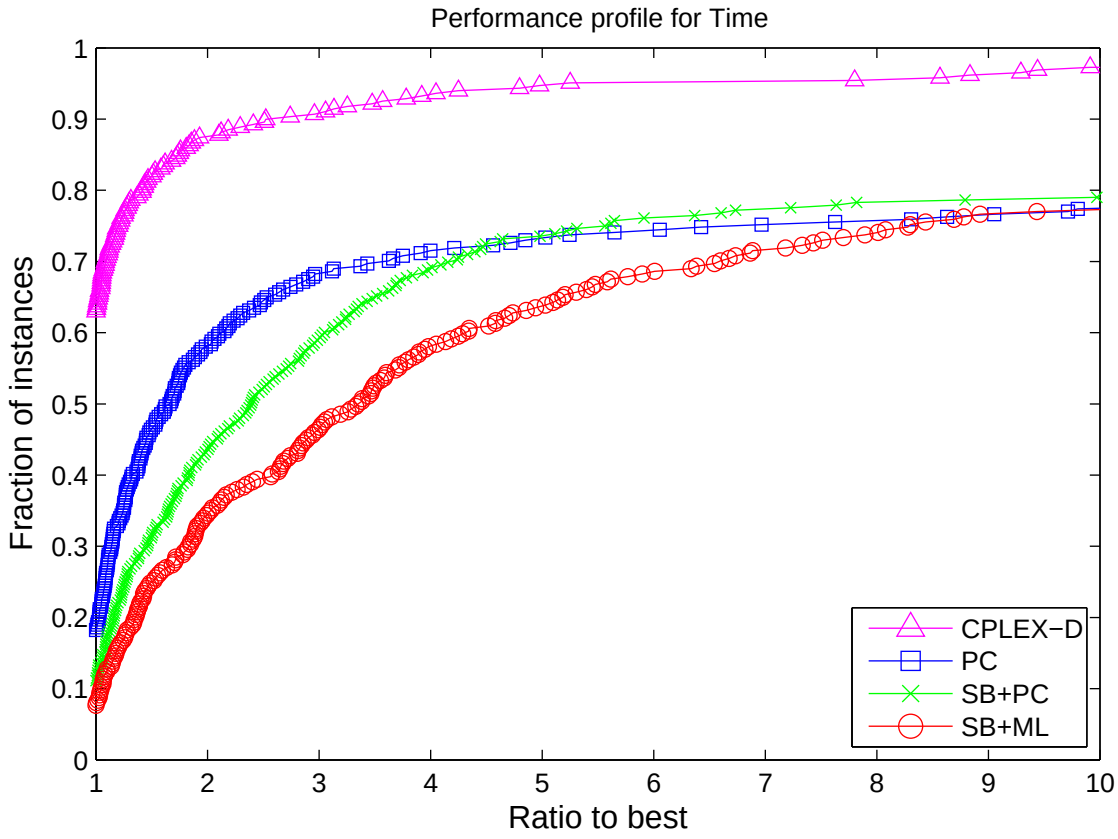


Figure 2.3: Performance profile for the total time. For a given strategy, a point (x, y) on this graph is the fraction y of instances solved by that strategy within a factor of x times more time than the best strategy, for each instance.

are uncorrelated. For our 1378 pairs of models, the mean correlation coefficient is 0.2 with a standard deviation of 0.14, indicating that only a very weak positive correlation exists between models, on average. A box plot for the distribution of the coefficients is shown in Figure 2.4. Interestingly, only 14 out of 1378 pairs of models exhibit a correlation of more than 0.5. This analysis seems to confirm our intuition: no “one-size-fits-all" rule for variable selection exists, and input-specific ranking models are discovered by the machine learning algorithm.

Top features. What are the features that are consistently given large (absolute) weights across different instances? To answer this question, we consider the same 53 models as before. For each model, we rank the features by their absolute weight. Then, for each feature,

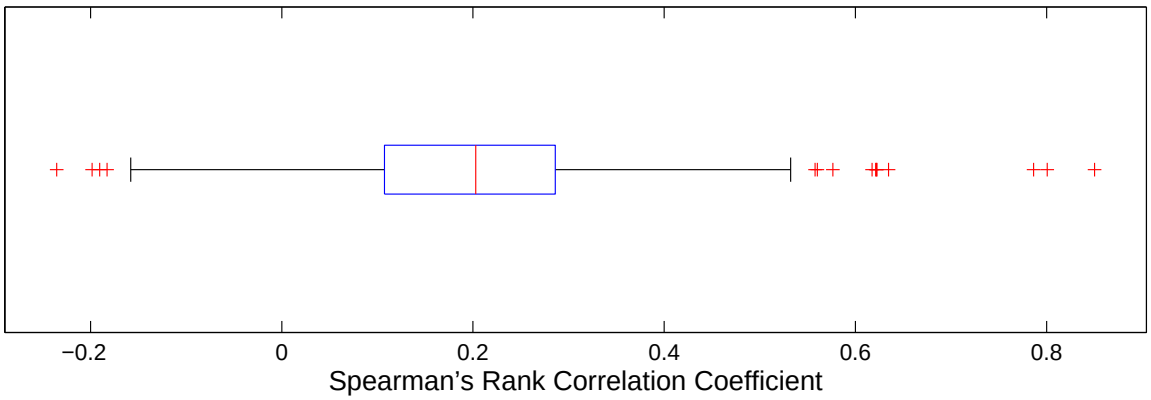


Figure 2.4: Box plot for the distribution of Spearman's rank correlation coefficients for 1378 pairs of models. "The central mark is the median, the edges of the box are the 25th and 75th percentiles, the whiskers extend to the most extreme data points not considered outliers, and outliers are plotted individually" [94].

we count the number of models in which it appeared among the top K ranked features. The 10 features that appear the most in the top 10 over models are presented in Table 3.2, along with counts for other values of K. Clearly, "PC Product" (PCP) is an important feature that is ranked in the top 10 in more than half of the models (27/53). However, that same feature, which is used on its own in PC branching, is ranked as the top feature only 3 times. This shows that combining various features does indeed bring about ranking models that are more predictive. Besides the PC Product and its square, the other eight features combine PCP with other static and dynamic features. Those include features related to the distribution of the number of variables in the constraints of the variable of interest, its coefficients in the constraints, and its coefficients in active constraints only (weighted by the inverse of the sum of the candidate variables' coefficients). These combinations are discovered to be important when learning the model, a task that is difficult to accomplish a priori, without data collection and learning.

2.8 Conclusions and Future Directions

We have proposed the first successful ML framework for variable selection in MIP, an approach which may also benefit other components of the MIP solver such as cutting

Table 2.7: The 10 features that appear the most in the top $K = 10$ features over models, sorted by that count (corresponding column is bold in the header). The counts for other values of K are also shown. There are 1385 features in total. Static features are marked by (S); unmarked features are dynamic.

Feature	1	3	5	**10**	20	100
PC Product	3	13	19	27	28	39
PC Product x Min. Constraint Degree	0	2	4	15	20	34
PC Product x Min. Positive Constraint Coefficient (S)	1	1	7	14	21	36
PC Product x Max. Constraint Degree	0	0	2	13	18	37
PC Product x Min. Negative Constraint Coefficient (S)	0	2	6	13	22	39
PC Product x Num. Constraints for Variable (S)	0	3	6	12	14	30
PC Product x Max. Absolute Coefficient in Active Constraints	1	4	6	12	21	37
PC Product x PC Product	4	4	5	11	19	35
PC Product x Mean Positive Constraint Coefficient (S)	0	5	6	11	14	34
PC Product x Mean Negative Constraint Coefficient (S)	1	2	5	11	19	36

planes and node selection. The framework can be extended in several directions, such as dynamically adjusting the number of training nodes θ or learning models multiple times in adaptation to the search progress. Beyond the batch supervised ranking approach we used, online and reinforcement learning formulations may be interesting to explore, given the structured, sequential nature of the variable selection task.

CHAPTER 3
LEARNING TO RUN HEURISTICS

"Primal heuristics" are a key contributor to the improved performance of exact branch-and-bound solvers for combinatorial optimization and integer programming. However, incorporating heuristics within tree search motivates challenging questions, the most important of which being: should the heuristic be run at a given node? Typical approaches to addressing this question involve an offline trial-and-error process, resulting in a set of hard-coded rules or fixed solver parameters. Alternatively, a heuristic should be run when it is most likely to succeed, based on the problem instance's characteristics, the state of the search, etc. In this work, we study the problem of deciding at which node a heuristic should be run, such that the overall (primal) performance of the solver is optimized. To our knowledge, this is the first attempt at formalizing and systematically addressing this problem. Central to our approach is the use of Machine Learning (ML) for predicting whether a heuristic will succeed at a given node. We give a theoretical framework for analyzing this decision-making process in a simplified setting, propose a ML approach for modeling heuristic success likelihood, and design practical rules that leverage the ML models to dynamically decide whether to run a heuristic at each node of the search tree. Experimentally, our approach improves the primal performance of a state-of-the-art Mixed Integer Programming solver by up to 6% on a set of benchmark instances, and by up to 60% on a family of hard Independent Set instances.

3.1 Introduction

The Primal Side of Integer Programming. There are two sides to any constrained optimization problem. On the one hand, we want to find *feasible solutions* to the problem instance at hand. On the other hand, we would like to *prove the optimality* of the best

feasible solution found, i.e. to guarantee that no feasible solution with strictly better objective function value exists. These two sides are particularly prominent in MIP, the study of optimization problems with integer-valued variables. To use the terminology of MIP, the *primal* side refers to the quest for good feasible solutions, whereas the *dual* side refers to the search for a proof of optimality.

While proving optimality is a key trait of exact solvers for MIP, finding quality feasible solutions quickly is certainly at least as crucial. For example, consider a real-world MIP model that a company solves on a regular (e.g. daily) basis in order to plan its operations. When state-of-the-art MIP solvers require many hours to solve an instance to optimality, the user will expect good feasible solutions to be found much earlier in the solving process, so that they are able to act upon them and address their business needs promptly. An example of such a challenging real-world scenario is that of the maritime inventory routing problem (MIRP), described in [104]. For the 28 MIRP instances, the solver Gurobi, with default settings (including parallel processing) and no warm-starting, is not capable of finding *any* feasible solutions for any of the instances in 24 hours [104]. The delay in finding feasible solutions affects the decision-maker's ability to plan ahead and compare options before deployment.

The Impact of Primal Heuristics. In this work, we focus on the primal side of integer programming. We do note, however, that finding better feasible solutions while solving a MIP with branch-and-bound speeds up proving optimality by pruning nodes with worse lower bounds, assuming a minimization problem. The classical way by which the MIP solver finds feasible solutions is through linear programming (LP) relaxations: in a branch-and-bound search, after branching on a subset of the integer variables of a MIP instance, solving the LP relaxation of the restricted sub-problem may result in an integer-feasible solution.

However, the MIP community has recently realized the potential for combining *primal heuristics* with exact branch-and-bound search to improve solution finding. Primal heuristics

are incomplete, bounded-time procedures that attempt to find a good feasible solution. Primal heuristics may be used as standalone methods, taking in a MIP instance as input, and attempting to find good feasible solutions, or as sub-routines inside branch-and-bound, where they are called periodically during the search. In this work, we focus on the latter, which we will expand on in the following paragraph.

A number of computational studies, with different MIP solvers, have demonstrated the large impact that primal heuristics have on branch-and-bound. An interesting finding reported in [22] is that on 97 easy benchmark instances, the LP relaxation finds an optimal solution 59 times, whereas on 26 hard instances it finds an optimal solution only 3 times; for the remaining instances, one of the primal heuristics of the SCIP solver used by Berthold finds an optimal solution. Berthold's results show that the investment in developing effective primal heuristics has brought about significant returns, most notably for harder instances. Even stronger results confirming the impact of heuristics on the solver CPLEX are discussed in [7].

Our Problem Setting. Despite the success of primal heuristics within MIP solving, there remains a number of central issues pertaining to *when* and *what* heuristics should be run during the search. For instance, in SCIP (a state-of-the-art academic MIP solver), 43 primal heuristics have been implemented. In the default settings of the solver, some heuristics are turned off, others run very frequently (e.g. at every node), while yet another subset runs occasionally (e.g. every 10 or 20 nodes). Such rigid rules for running heuristics are static, instance-oblivious, context-independent, and are unable to adapt to the state of the search. Additionally, the algorithmic differences between primal heuristics result in substantial variation in performance. For instance, diving and neighborhood search heuristics are much more computationally expensive than their rounding counterparts, but are generally more likely to find quality feasible solutions.

Towards establishing a *dynamic, data-driven approach* to the use of primal heuristics in tree search, we address the problem of decision-making for primal heuristics. Assume

that $P(t)$ is some primal performance measure, whose value at time point t indicates how successful the solver has been on the primal side up to t; the choice of the performance measure $P(.)$ will be discussed in detail in Section 3.3. In its simplest form, a formulation of the problem addressed here is:

> Given a primal heuristic H, a branch-and-bound solver with search tree $\mathcal{T}$, a time cutoff t_{max}, find the subset of nodes of $\mathcal{T}$ at which executing H results in the best primal performance possible, $P(t_{max})$.

To our knowledge, the systematic study of this problem is new. By "systematic", we mean that there is a well-defined *objective function* $P(t_{max})$ to optimize, and a clear *decision space*, namely executing H or not at each node. We refer to a procedure that decides when to run a primal heuristic as a *primal policy*. The proposed problem raises a number of interesting questions that span online decision-making under uncertainty and ML.

3.2 Primal Heuristics

In order to incorporate a primal heuristic within the branch-and-bound framework, the developer or user of a MIP solver must make certain decisions that reflect their belief in the heuristic's potential for finding high-quality feasible solutions efficiently. Most importantly, one must decide the *frequency*, in terms of number of nodes, with which the heuristic will be run. Additionally, internal parameters of the heuristic must be set to suitable values. To make these decisions, one must understand the current landscape of primal heuristics, which we now briefly describe.

Functionally, primal heuristics can be categorized into *start heuristics*, which aim at finding a *first* feasible solution, and *improvement heuristics*, which aim at producing new, better feasible solutions. Algorithmically, primal heuristics fall into three broad categories: diving, rounding and propagation, and large neighborhood search. Diving heuristics simulate a depth-first traversal from a given node, and are motivated by the assumption that feasible

solutions are more likely to come about in deeper levels of the search tree. Rounding and propagation heuristics use the LP relaxation solution at a node as input, and try to turn it into an integral solution with simple transformations. As for neighborhood search heuristics, they typically solve a sub-MIP problem that models the neighborhood of existing feasible solutions and/or the LP solution at a node, with the hope of finding a better feasible solution in that neighborhood. These algorithmic differences result in primal heuristics that exhibit substantial variation in performance. For instance, diving and neighborhood search heuristics are substantially more computationally expensive than their rounding counterparts, but are generally more likely to find quality feasible solutions.

3.3 The Primal Integral

Our goal is to improve the primal performance of tree search, i.e. the quality of and the speed at which feasible solutions are found. The *primal integral* is a primal performance criterion for MIP that was introduced in [5] to formally capture these desired characteristics, and that we adopt as a main measure of primal performance.

Let x^* denote an optimal (or best known) solution for a MIP, and $\tilde{x}$ denote a feasible solution for the same MIP. The *primal gap* $\gamma \in [0, 1]$ of solution $\tilde{x}$ is defined as:

$$\gamma(\tilde{x}) \coloneqq \begin{cases} 0, & \text{if } |c^T x^*| = |c^T \tilde{x}| = 0 \\ 1, & \text{if } c^T x^* \cdot c^T \tilde{x} < 0 \\ \frac{|c^T \tilde{x} - c^T x^*|}{\max\{|c^T \tilde{x}|, |c^T x^*|\}}, & \text{otherwise.} \end{cases} \tag{3.1}$$

Let $t_{\text{max}} \in \mathbb{R}_{\geq 0}$ be a limit on the solution time of the B&B MIP solver. Then, the *primal gap function* $p : [0, t_{\text{max}}] \mapsto [0, 1]$ is defined as:

$$p(t) \coloneqq \begin{cases} 1, & \text{if no incumbent is found until point } t, \\ \gamma(\tilde{x}(t)), & \text{with } \tilde{x}(t) \text{ the incumbent at point } t. \end{cases}$$

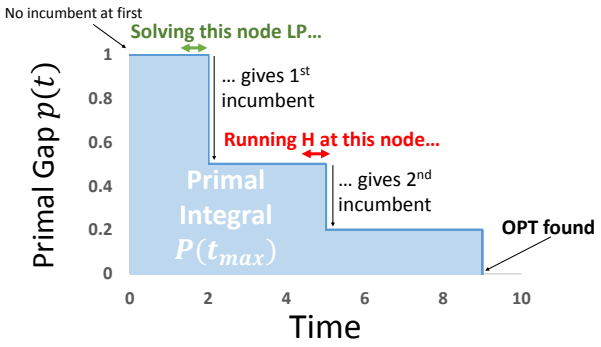


Figure 3.1: An illustration of the primal integral.

It is easy to see that $p(t)$ is a nonincreasing step function in $[0, 1]$ that changes whenever a new incumbent is found, and takes on a value of zero the moment an optimal solution is found – see Figure 3.1. The *primal integral* $P(T)$ of a branch-and-bound run until a point in time $T \in [0, t_{\text{max}}]$ is defined as:

$$P(T) := \sum_{i=1}^{Inc+1} p(t_{i-1}) \cdot (t_i - t_{i-1}),$$

where Inc is the number of incumbents, $t_i \in [0, T]$ for $i \in 1, \ldots, Inc$ are the points in time when a new incumbent is found, $t_0 = 0$ and $t_{Inc+1} = T$. A graphical illustration of the primal integral is shown in Figure 3.1. Note that the *primal-dual integral* is another metric that is defined similarly to the primal integral, with $\gamma(\tilde{x}, \underline{z}) := \frac{c^T \tilde{x} - \underline{z}}{\max\{|c^T \tilde{x}|, |\underline{z}|\}}$, $PD(T) := \sum_{i=1}^{Chn} p(t_{i-1}) \cdot (t_i - t_{i-1})$, Chn the time points at which either the global lower bound $\underline{z}$ or upper bound $c^T \tilde{x}$ changed, and $p(t_{i-1}) = \gamma(\tilde{x}(t), \underline{z}(t))$ (or 1 if either bound is undefined). However, the primal-dual integral confounds the primal and dual sides, and is thus less relevant for our purposes.

Achterberg et al. suggest that the primal integral be used to measure the progress on the primal side during B&B [5]. The smaller $P(t_{\text{max}})$ is, the better the incumbent finding. As such, we will consider optimizing the *primal integral* directly, by means of making good decisions regarding whether a primal heuristic should be run at each node or not.

3.4 Theoretical Analysis

3.4.1 Problem Formulation

Should heuristic H be run at a given node N? An answer to this question must study the trade-off between the potential benefit of finding a better feasible solution if H is run at N, and the cost associated with running H (including the risk of failure). Running H at every node may be undesirable, as H may run for a long period of time, during which the MIP could be solved to optimality, irrespective of H. Thus, it is crucial to choose the right set of nodes at which to run H. We now give the first general formulation of the problem we call "Primal Integral Optimization" (PIO):

> **(PIO)** Given a primal heuristic H, a branch-and-bound MIP solver with search tree $\mathcal{T}$ and a time cut-off t_{max}, find the subset of nodes of $\mathcal{T}$ at which executing H results in a primal integral $P(t_{max})$ of minimum value.

The first step towards formalizing PIO lies in defining a simple, conceptual model of branch-and-bound, within which we can analyze the complexity of PIO, and the theoretical performance of approaches to solving it. We will distinguish two main settings:

- the *offline* setting, where the search tree $\mathcal{T}$ is *fixed and known* in advance, and PIO amounts to finding the best subset of nodes to run H at in hindsight;
- the *online* setting, where one must sequentially decide, at each node, whether H should be run, without any knowledge of the remainder of the tree or search.

In practice, the online setting is more relevant as it is representative of actual MIP solving. Thus, we will analyze online decision-making algorithms next, and bound their worst-case performance compared to an optimal solution obtained offline, in hindsight. Note that such an offline solution is easy to compute via dynamic programming if the B&B tree is known and fixed in advance. We do not provide the details of the offline algorithm, as it is not of practical interest.

Central to the algorithms that we analyze is the notion of an *oracle* which, when queried at a given node, tells the online algorithm whether heuristic H will find an incumbent or not. We will analyze the performance of an online algorithm under two different assumptions on the oracle's behavior. In the first setting, the oracle is assumed to be *perfect*, in that it returns the correct answer (i.e. whether H will find an incumbent or not) at every node at which it is queried. In the second setting, the oracle is assumed to be *faulty*, making a mistake with a given probability.

To see how this conceptual framework is tied to the proposed ML approach to PIO, notice that an ML model of heuristic success can be seen as a faulty oracle: the model is likely to miss a few incumbents, or wrongly predict incumbents at some nodes. Since the heuristic is run at nodes during B&B, any decision-making algorithm must act online, using only information about the solving process up to the given point in time. Our theoretical analysis aims at substantiating the practical ML approach to PIO of Section 3.5.

3.4.2 Competitive Ratio under a Perfect Oracle

The online PIO problem is a challenging one, since decisions regarding running H at a node must be made without any knowledge of the remainder of the search tree, and the incumbents that may be found later by LP or H. As a first result, we will analyze the following simple rule of thumb for deciding whether to run a primal heuristic at a node: if the oracle says that H can find an incumbent solution at node N_i, then run H; otherwise, do not run H. We refer to this rule of thumb as Run-When-Successful (RWS).

To analyze online algorithms, we resort to worst-case analysis using the *competitive ratio* [9]. Assume we are given a sequence of "requests" σ (in our case, each request is a node at which we run H or not). Let $A(\sigma)$ denote the cost incurred by a deterministic online algorithm A, and let $OPT(\sigma)$ denote the cost incurred by an optimal offline algorithm OPT. The algorithm A is called c-competitive if there exists a constant a such that $A(\sigma) \leq c \cdot OPT(\sigma) + a$, for $\forall\sigma$; c is the *competitive ratio*.

Theorem 1 *RWS achieves a competitive ratio of*

$$1 + \frac{t_H}{t_{LP}}$$

with respect to the optimal offline solution, where t_H, t_{LP} are the fixed running times of H and an LP relaxation solve, respectively.

PROOF. Let P^{RWS} denote the primal integral value obtained on an instance I using the RWS rule, and P^{OPT} the optimal primal integral value for I. Assume the optimal primal policy for heuristic H goes through n nodes before finding an optimal solution to the MIP I. Notice that RWS will require at most n nodes as well, since RWS guarantees that at any node, its incumbent is the best possible up to that node. Let $\mathcal{T}_H^{RWS}$ be the set of time points at which RWS runs H, and $\mathcal{T}_{LP}^{OPT}$ the set of time points at which the optimal primal policy solves an LP. Then, P^{RWS} can be upper bounded as:

$$P^{RWS} \leq P^{OPT} + \sum_{t_i \in \mathcal{T}_H^{RWS}} p(t_i) \cdot t_H.$$

The upper bound is valid because the worst RWS can do is run for t_H seconds multiple times and not improve the incumbent. Dividing both sides of the above inequality by P^{OPT}, we obtain:

$$\begin{aligned} \frac{P^{RWS}}{P^{OPT}} &\leq 1 + \frac{\sum_{t_i \in \mathcal{T}_H^{RWS}} p(t_i) \cdot t_H}{P^{OPT}} \\ &\leq 1 + \frac{\sum_{t_i \in \mathcal{T}_H^{RWS}} p(t_i) \cdot t_H}{\sum_{t_i \in \mathcal{T}_{LP}^{OPT}} p(t_i) \cdot t_{LP}} \leq 1 + \frac{t_H}{t_{LP}}. \end{aligned}$$

The second inequality is valid because

$$P^{OPT} \geq \sum_{t_i \in \mathcal{T}_{LP}^{OPT}} p(t_i) \cdot t_{LP},$$

i.e. the optimal primal integral has value at least that of the LP solves weighted by the primal

gap $p(t_i)$. The final inequality is valid because

$$\sum_{t_i \in \mathcal{T}_H^{RWS}} p(t_i) \cdot t_H \leq \sum_{t_i \in \mathcal{T}_{LP}^{OPT}} p(t_i) \cdot t_{LP},$$

as H is run at most as frequently as LPs are solved, and the primal gap value at any node at which RWS runs H is at worst equal to the corresponding gap value for the optimal primal policy. □

It is interesting to see how the bound in this theorem holds up in practice on real MIP instances from the Benchmark set of MIPLIB2010 [81]. Table 3.1 shows the empirical competitive ratio of the RWS algorithm and the theoretical one, computed as in Theorem 1, for four sample instance-heuristic pairs. The optimal offline primal integral P^{OPT} is computed via dynamic programming in hindsight. The results of Table 3.1 show that the empirical performance of RWS is much better than the theory suggests. However, we do believe that the bound is tight up to an arbitrarily small constant.

Table 3.1: Sample results from empirical evaluation of RWS.

Instance	Heuristic	Empirical $\frac{\hat{P}^{RWS}}{P^{OPT}}$	Theoretical $\frac{P^{RWS}}{P^{OPT}}$
biella1	fracdiving	1.09	122.79
rail507	veclendiving	1.09	18.36
qiu	guideddiving	1.01	5.32
biella1	intshifting	1.00	3.57

3.4.3 Competitive Ratio under a Faulty Oracle

Any practical oracle, such as one designed with ML, will not be perfect. A *faulty* oracle is one that makes a *mistake* at a node with some probability. We distinguish two types of mistakes: false positives and false negatives. Assume that the oracle incurs a false positive at a node with probability δ, i.e. the oracle states that H will find an incumbent at a node N when H does not, and a false negative with probability β, i.e. the oracle states that H will not find an incumbent at N when H does. When $\delta > 0, \beta = 0$, the bound from Theorem 1 also

holds for the competitive ratio w.r.t. the *expected* primal integral, i.e. $\frac{\mathbb{E}[P^{RWS}]}{P^{OPT}} \leq 1 + \frac{t_H}{t_{LP}}$. The randomness is with respect to the nodes at which the false positives occur. Unfortunately, when $\beta > 0$, bounding the competitive ratio becomes much trickier: if H finds an optimal solution at N which cannot be found at any other node by either H or LP relaxation solves, and a false negative occurs at N, then $P^{RWS} = t_{max}$, the maximum possible value. As such, we believe that any such bound when $\beta > 0$ will be very loose. However, there may be suitable assumptions under which the bound is not as loose, and we consider this issue to be interesting for future research.

3.5 Learning a Success Oracle for Heuristics

In the previous section, we showed that despite its simplicity, the RWS rule provides theoretical guarantees under a simplified setting. However, in order to turn RWS into an operational policy, we must design a success oracle. Combining the designed oracle with RWS provides an online procedure for deciding when to run a heuristic during tree search. Recall that our aim is to *dynamically* decide whether to run the heuristic at a given node, based on the instance characteristics, node characteristics and state of the search. As such, we will design the oracle by *learning* a binary classifier which predicts whether heuristic H will find an incumbent solution at node N. The features used to describe node N will incorporate information about the instance, the node and the state of the search, as desired. The proposed framework is illustrated in Figure 3.2.

3.5.1 Realistic Data Collection

We now describe our method for collecting data on the heuristic's success across different instances. This aspect of oracle design warrants special attention, given the interplay between incumbent finding and tree search. More specifically, one has to make sure that the node data collected for heuristic H for training is similar to the node data to be encountered when using H's oracle, online, on a new problem instance.

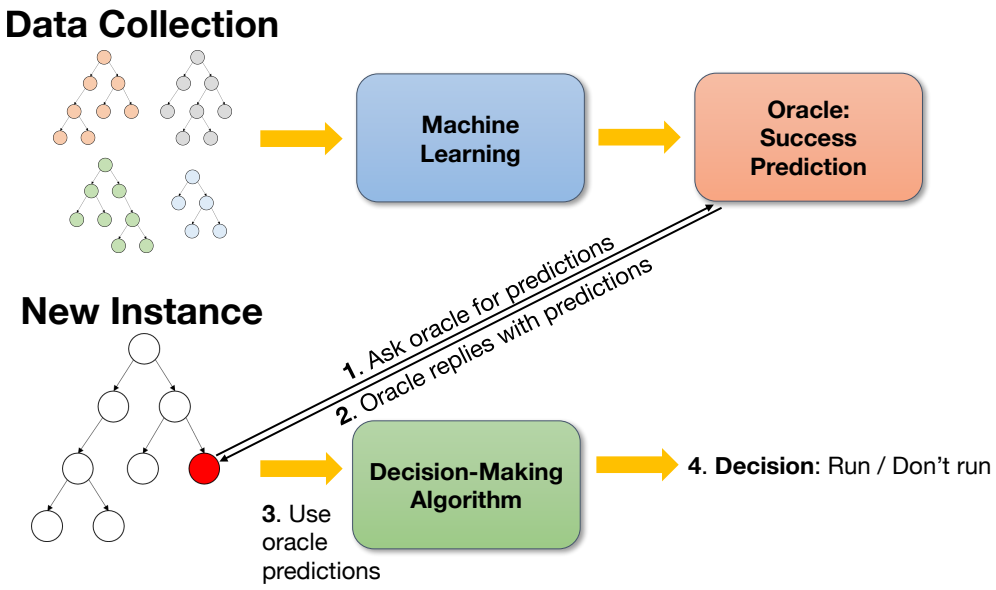


Figure 3.2: An illustration of the proposed framework for the Primal Integral Optimization problem. The top part relates to the training process, whereas the bottom part shows the framework at test time.

Let $\mathcal{H}$ be the set of primal heuristics used during tree search, $H \in \mathcal{H}$ a given heuristic, and $\bar{\mathcal{H}} = \mathcal{H} - H$. We are given a set of MIP instances, $\mathcal{I}_{train}$, which can be used to collect data on H. A dataset D_I^H is obtained for each instance $I \in \mathcal{I}_{train}$, and the final training dataset for heuristic H is obtained by concatenating instance datasets together into $\mathcal{D}_{train}^H = \bigcup_{I \in \mathcal{I}_{train}} D_I^H$.

We now consider data collection at the individual instance level. Let $I \in \mathcal{I}_{train}$ be a MIP instance for which we want to collect data for heuristic H. We will run H at every node N of the search tree, and collect the binary classification label value, $y_H^N \in \{0, 1\}$, and the feature vector $\mathbf{x}^N \in \mathbb{R}^d$. The label y_H^N takes a value of 1 if H finds an incumbent at node N, and 0 otherwise. The key observation here is that the value of y_H^N depends on z_N^*, the objective function value of the incumbent when node N is considered. In turn, the value of z_N^* depends on the progress in the search up to N. If H is run at every node, then any incumbent that H finds affects the value $y_H^{N'}$ for all nodes N' that come after N. This interplay between the incumbents found by H and the data being collected for H is problematic, as the labels y_H^N in the training dataset assume that the oracle is perfect, i.e. H

is run anytime it can find an incumbent. In practice, however, the oracle is a binary classifier that is unlikely to be perfect, meaning that it will not always run H, even when H can find an incumbent.

To deal with this complication, we devise a data collection procedure that is more likely to result in realistic datasets. First, for any heuristic H for which data is to be collected, H is run in "stealth" mode: any new incumbent that H finds does not replace the current incumbent. This measure is equivalent to not running H at all from the branch-and-bound perspective, while still obtaining useful data for H. Second, all other heuristics $H' \in \bar{\mathcal{H}}$ are run using their default solver frequencies, which simulates actual MIP solving, where many heuristics are interacting together.

3.5.2 Designing Node Features

So far, we have discussed collecting data that is realistic w.r.t. the target label. We now discuss the choice of features used to describe a given node N. We use a d-dimensional feature vector, $\mathbf{x}^N \in \mathbb{R}^d$, with $d = 49$, consisting of the features listed in Table 3.2. *Global features* describe the current state of the search using gap-related metrics. The (optimality) gap is defined as the relative difference between the global upper bound (i.e. the objective value of the best incumbent so far) and the global lower bound (i.e. the best possible objective value, due to LP relaxations). *Node LP features* use the solution of the LP relaxation at a node N to obtain certain indicative metrics. For instance, the feature "Num. of Active Constraints / Num. of Constraints" can be indicative of how sensitive the LP is to the fixing of additional variables, which is important for diving heuristics (an active constraint is one that is satisfied with equality at the LP solution). *Scoring Features for Fractional Variables* are inspired by the scoring functions that various diving heuristics use to select the next variable to fix. Details on the definitions of these functions are given in section 1.4.2 of [57]. Essentially, for a given scoring function f : fractional variables $\rightarrow \mathbb{R}$, we compute the value of f for each fractional variable in the node's LP solution, compute statistics over the f

values, and use those as features.

One trait of our features is that they are *naturally scaled*, i.e. each feature is appropriately divided by a scaling factor that depends only on the MIP instance (e.g. number of variables or constraints) or the node itself (e.g. number of fractional variables). Having appropriately scaled features is important for the convergence of many learning algorithms. However, that is not the only reason for emphasizing this aspect of our feature design. In fact, scaling is important in our setting because training data comes from multiple instances, each with its own dimensions, structure, etc. As such, the standard approach of scaling/normalizing data for training is not enough here: the scaling factors may not be directly applicable to a new instance's data. We have carefully crafted the features such that they can be scaled appropriately online, using only local information from the node and the instance.

3.6 Experimental Results

To evaluate the proposed framework, we modify the open-source MIP solver SCIP 3.2.1 [49]; CPLEX 12.6.1 [65] is used as SCIP's LP solver. Machine learning experiments use *scikit-learn* [107]. All experiments were run on a cluster of four 64-core machines with AMD 2.4 GHz processors and 264 GB RAM.

3.6.1 Oracle Learning

Heuristics. As a first phase, we learn a binary classifier that predicts incumbent success. We consider a set of ten heuristics implemented in SCIP, listed in the first column of Table 3.3. The ten heuristics were selected out of the 43 implemented in SCIP after excluding heuristics that are disabled by default (17), require a feasible solution (4), are too cheap (8), are for non-linear programs (3), or are for special constraints (1).

ML Setup. For a given heuristic H and a dataset $\mathcal{D}^H_{train} = \bigcup_{I \in \mathcal{I}_{train}} D^H_I$ collected from a set of training instances $\mathcal{I}_{train}$, we use *logistic regression* (LR) to learn a binary classification model, $\mathbf{w}_H \in \mathbb{R}^d$. The regularization parameter of LR is kept at a default of 1. Data

Table 3.2: List of the 49 features used. “Scoring features for fractional variables" are five statistics (mean, min., max., median, standard deviation) for each of seven metrics over fractional variables.

Global Features (4)
Optimality gap
Infinite gap?
Root LP value / Global Lower Bound
Root LP value / Global Upper Bound
Depth Features (2)
Node Depth / Max. Depth in Tree
Node Depth / Max. Possible Depth
Node LP Features (8)
Sum of variables’ LP solution fractionalities / Num. of Fractional Variables
Num. of Fractional Variable / Num. of Integer Variables
Num. Variables Roundable Up (Down) / Num. of Integer Variables (x2)
Num. of Active Constraints / Num. of Constraints
Node is root?
Root LP value / Node LP value
Root LP value / Node Estimate
Scoring Features for Fractional Variables (35)
Number of Up Locks (x5) – Number of Down Locks (x5)
Normalized Objective Coefficient (x5)
Objective Gain (x5)
Distance to root LP solution (x5)
Vector Length (x5)
Pseudocost score (x5)

Table 3.3: Leave-one-out cross-validation accuracy results for logistic regression on 10 primal heuristics in SCIP on MIPLIB2010 Benchmark. “AUC-ROC" is the area under the “receiver operating characteristic" curve. “Precision" is the fraction of points from the positive class out of all points classified as positive. “Recall" is the fraction of points from the positive class that are classified as positive. For both precision and recall, the results are using a threshold of 0.5 on the predicted probabilities.

Heuristic	Num. Instances	Num. Data Points	Success Rate	Precision		Recall		AUC-ROC	
				Mean +/- Std.	Median	Mean +/- Std.	Median	Mean +/- Std.	Median
coefdiving	44	2,635,296	0.0002	0.0264 +/- 0.0784	0.001	0.6552 +/- 0.3872	0.876	0.8543 +/- 0.1400	0.896
distributiondiving	51	2,721,704	0.0004	0.0255 +/- 0.0604	0.001	0.6715 +/- 0.3667	0.903	0.8075 +/- 0.1884	0.831
fracdiving	37	2,721,288	0.0001	0.0044 +/- 0.0093	0.001	0.6466 +/- 0.3810	0.688	0.7953 +/- 0.2184	0.878
intshifting	9	1,652,684	0.0001	0.1213 +/- 0.2291	0.001	0.4018 +/- 0.4347	0.177	0.8644 +/- 0.0823	0.865
linesearchdiving	34	2,552,685	0.0001	0.0170 +/- 0.0690	0.001	0.6794 +/- 0.3919	0.889	0.8187 +/- 0.1343	0.819
objpscostdiving	10	10,329	0.0127	0.3539 +/- 0.3814	0.131	0.5514 +/- 0.3769	0.486	0.8712 +/- 0.2247	0.996
pscostdiving	57	2,531,343	0.0007	0.0206 +/- 0.0430	0.002	0.6082 +/- 0.3636	0.716	0.7176 +/- 0.2359	0.773
rootsoldiving	6	6,047	0.0026	0.0990 +/- 0.1826	0	0.3333 +/- 0.4714	0	0.9599 +/- 0.0569	0.986
veclendiving	38	2,785,210	0.0002	0.0255 +/- 0.0687	0.003	0.7953 +/- 0.2799	0.936	0.7929 +/- 0.1923	0.829

points with label $y_H^N = 1$ are heavily weighted in the LR loss function to account for the extreme class imbalance we encounter [55], as can be seen in column “Success Rate" of Table 3.3. The model $\mathbf{w}_H$ is simply a weight vector for the $d = 49$ node features described in Section 3.5.2, such that the dot product $\langle \mathbf{w}_H, \mathbf{x}^N \rangle$ between $\mathbf{w}_H$ and node N's feature vector $\mathbf{x}^N$ gives an estimate of the probability that heuristic H finds an incumbent at N. We have experimented with other ML models that have more capacity (and hyper-parameters) (SVM, gradient boosted trees), but have not observed any benefit from such models in either classification performance or learning/prediction time.

ML Results. We use leave-one-out cross-validation (LOOCV) on a per-instance basis: for each test instance I_{test}, a model is learned for a heuristic H using dataset $\mathcal{D}^H_{train}$, where $\mathcal{D}^H_{train}$ does not include any data from I_{test}; the model is then tested on I_{test}'s dataset, $D^H_{I_{test}}$.

Table 3.3 shows LOOCV results using 83 instances of the “Benchmark" set from MIPLIB2010 [81], for which data was collected by running SCIP for 2 hours at most, per instance. First, observe that the datasets at hand are extremely imbalanced. For instance, the success rate (i.e. the fraction of nodes in the collected dataset for which the heuristic succeeds in finding an incumbent) of the *coefdiving* heuristic is 0.000192: only 1 in $5,000$ runs result in an incumbent. As such, the “Precision" of an ML success oracle must be better than random prediction (which succeeds with rate equal to the success rate). For each

of the ten heuristics, Table 3.3 shows the average precision, recall and AUC-ROC, over instance datasets with at least one positive label data point (otherwise, these metrics are undefined). Fortunately, the average precision of the learned models is orders of magnitude better than the success rate: for *coefdiving*, the ML model is more than 100 times more precise in classifying incumbents than at random. Additionally, the recall of the models is satisfactory, with most incumbents being detected for most heuristics.

Despite the heterogeneous nature of the instances, our framework is able to learn oracle success models that are significantly better than random guessing, despite extreme class imbalance. Next, we study the impact of using the learned oracles, in conjunction with the RWS rule, on the solver's primal performance.

3.6.2 MIP Solving

Setup. While the ML results for the success oracles are positive, they are only of practical use if they can improve the performance of a state-of-the-art MIP solver w.r.t. primal metrics such as the primal integral. We use the learned oracles in conjunction with the Run-When-Successful rule to guide the decisions as to whether each of the ten heuristics of Table 3.3 should be run at each node. Specifically, for a given instance, the ten heuristics' models are loaded, and used to compute the probability of success of a heuristic given a node's feature vector. For other heuristics without ML oracles, we use their default settings in SCIP.

We compare our approach, referred to as ML, with the solver's default policy, DEF. For each of the 83 MIPLIB2010 "Benchmark" set instances, we run every policy with 5 different random permutations of the rows and columns of the instance; each instance-permutation pair is considered as a separate instance. This measure is a standard one for computational MIP studies, as it helps to control for the inherent "performance variability" in solvers – see [91, 7] for details.

MIP Results. Table 3.4 (left) summarizes the results. Table 3.4 (left) shows that our

Table 3.4: Summary of results on the MIPLIB2010 Benchmark set with 5 random permutations per instance (Top), and the GISP test set (Bottom); $t_{max} = 7,200$. For MIPLIB2010, instances requiring less than 10 minutes for either DEF or ML are excluded as too easy. Values shown are aggregates over instances: geometric means are used for all but *Num. Instances Solved* (count), *Num. incumbents*, *Success rate* and *Num. incs. per heur. sec.* (arithmetic means). For GISP, the *Primal integral* uses the best upper bound rather than the optimal solution.

MIPLIB – Num. Instances = 280	**DEF**	**ML**	**ML/DEF**
Primal integral	95.65	89.65	0.94
Time to first incumbent	34.23	26.60	0.78
Time to best incumbent	746.95	738.71	0.99
Total calls (ML heurs.)	755.19	514.77	0.68
Total time (ML heurs.)	124.38	101.88	0.82
Num. incumbents (ML heurs.)	1.85	2.45	1.33
Success Rate (ML heurs.)	0.00036	0.00064	1.79
Num. incs. per heur. sec. (ML heurs.)	0.00565	0.00860	1.52
Num. Instances Solved	170	172	1.01
Total time (BnB)	3,966.47	4,119.67	1.04
Total nodes (BnB)	27,458.77	27,904.43	1.02
Primal-dual integral	34,390.33	35,329.91	1.03

GISP – Num. Instances = 120	**DEF**	**ML**	**ML/DEF**
Primal integral	2,621.79	1,038.58	0.40
Time to first incumbent	0.19	0.19	1.00
Time to best incumbent	5,601.44	2,166.98	0.39
Total calls (ML heurs.)	49.37	63.59	1.29
Total time (ML heurs.)	194.42	610.64	3.14
Num. incumbents (ML heurs.)	1.48	2.69	1.82
Success Rate (ML heurs.)	0.02566	0.03710	1.45
Num. incs. per heur. sec. (ML heurs.)	0.00501	0.00319	0.64
Num. Instances Solved (% Gap)	0 (201.95)	0 (181.35)	N/A (0.90)
Total time (BnB)	7,200.00	7,200.00	1.00
Total nodes (BnB)	619.19	476.94	0.77
Primal-dual integral	520,674.41	454,162.12	0.87

framework, ML, results in a reduction of 6% in the primal integral. Similarly, the time to the first and best incumbents are both improved by 22% and 1%, respectively. This is despite having an extremely heterogeneous set of training and testing instances. Our method makes better use of the heuristics it controls, as shown by the second set of rows in Table 3.4 (left): fewer calls are made to the ten heuristics, yet more incumbents are found by them on average, compared to DEF. Most notably, the success rate of ML-controlled heuristics is 1.79 times larger than that of DEF, and the number of incumbents found per second is 1.52 times larger. These figures, over a large set of benchmark instances, support our hypothesis: *dynamic decision-making for heuristics using the proposed framework improves the primal performance of an optimized state-of-the-art solver*.

3.6.3 Generalized Independent Set Problem

The experiments presented so far are on a heterogeneous set of MIP instances. However, in many real-world settings, one solves the same *homogeneous* family of problems, where instances differ only slightly in the number of constraints or variables, while maintaining the same overall combinatorial structure. To assess the effectiveness of our framework on a homogeneous instance set, we perform the same oracle learning and MIP solving experiments on instances of the *Generalized Independent Set Problem* (GISP) [60, 31].

Recently, it has been shown that the GISP requires specialized techniques to obtain good feasible solutions [31], which motivated our choice of this problem. Given a graph $G(V, E)$, a subset of removable edges $E' \subseteq E$, revenues for each vertex and costs for each removable edge, GISP asks to select a subset of the vertices and a subset of removable edges that maximize the profit, i.e. the difference between selected vertex revenues and removable edge costs. No edge should exist between any two selected vertices u and v, i.e. $(u, v) \notin E$, or $(u, v) \in E'$ and (u, v) is removed.

We use the twelve graphs from the 1993 DIMACS Challenge [70], also used in [31]. Six

instances [1] are held out for data collection and training, and six others [2] for MIP testing. The graphs are dense, with training graphs having 125 – 300 nodes and 6963 – 20864 edges, testing graphs having 250 – 400 nodes and 21928 – 71819 edges. For each of the twelve graphs, we generate 20 GISP instances by randomizing the set of removable edges, as in [31]: each edge is in the set E' with probability α. We use $\alpha = 0.75$, each node has revenue 100 and each removable edge has cost 1, a configuration shown to be difficult w.r.t. finding feasible solutions (SET2-A in [31]). Note that, even for the same graph, its 20 instances have a different number of variables for removable edges and different constraints.

We collect data for eight diving heuristics (the heuristics listed in Table 3.3 except feaspump and intshifting, which SCIP did not run), and learn corresponding oracles. Then, we test the oracles on the 120 test instances that were not seen during learning. The primal integral requires an optimal or best integer solution, for which we use the best solution found by multi-threaded CPLEX 12.6.1 after 10 hours of solving. The results are shown in Table 3.4 (right).

A dramatic improvement in the primal integral can be observed, with ML costing only 0.4 of DEF. This improvement can be largely attributed to the reduction in the time to best incumbent, also down to 0.39 of DEF: ML needs around 1 hour less than DEF to find its best incumbent, over a time cutoff of 2 hours. As for the quality of the best incumbent, ML finds a better one than DEF in 93 of 120 of the instances (77%). For all 120 instances, ML has a better primal integral than DEF.

The larger reduction in the primal integral of GISP instances, as compared to the MIPLIB2010 Benchmark set, is consistent with the intuition that learning on homogeneous instances is easier than on heterogeneous ones. Note that the GISP training instances had fewer variables and constraints due to the smaller graphs, yet the classifiers were very effective on the larger test instances, indicating that generalization on homogeneous instance sets is possible.

[1] C125.9,keller4,brock200_2,p_hat300-1,gen200_p0.9_55,hamming8-4

[2] p_hat300-2, C250.9, p_hat300-3, brock400_2, MANN_a27, gen400_p0.9_75

3.7 Conclusions and Future Work

We have shown that intelligent decision-making for heuristics can boost the performance of a state-of-the-art optimization solver, even on instances for which the solver is already fine-tuned by experts. To our knowledge, this work represents the first systematic attempt at optimizing the use of heuristics in tree search. This work lays the ground for fruitful future extensions, such as more refined rules that take into account the running time of the heuristics and the amount of time remaining for the solver, approaches for more effective scheduling of heuristics within a node, and online learning of good rules.

Part II

Learning Heuristics for Discrete Optimization

CHAPTER 4
GREEDY GRAPH OPTIMIZATION

The design of good heuristics or approximation algorithms for NP-hard combinatorial optimization problems often requires significant specialized knowledge and trial-and-error. Can we automate this challenging, tedious process, and learn the algorithms instead? In many real-world applications, it is typically the case that the same optimization problem is solved again and again on a regular basis, maintaining the same problem structure but differing in the data. This provides an opportunity for learning heuristic algorithms that exploit the structure of such recurring problems. In this chapter, we propose a unique combination of reinforcement learning and graph embedding to address this challenge. The learned greedy policy behaves like a meta-algorithm that incrementally constructs a solution, and the action is determined by the output of a graph embedding network capturing the current state of the solution. We show that our framework can be applied to a diverse range of optimization problems over graphs, and learns effective algorithms for the Minimum Vertex Cover, Maximum Cut and Traveling Salesman problems.

4.1 Introduction

Combinatorial optimization problems over graphs arising from numerous application domains, such as social networks, transportation, telecommunications and scheduling, are NP-hard, and have thus attracted considerable interest from the theory and algorithm design communities over the years. In fact, of Karp's 21 problems in the seminal paper on reducibility [71], 10 are decision versions of graph optimization problems, while most of the other 11 problems, such as set covering, can be naturally formulated on graphs. Traditional approaches to tackling an NP-hard graph optimization problem have three main flavors: exact algorithms, approximation algorithms and heuristics. Exact algorithms are

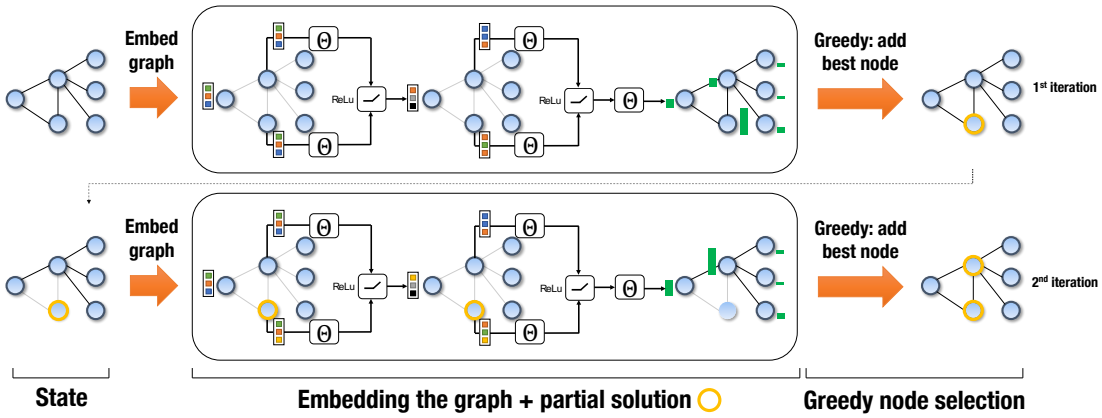


Figure 4.1: Illustration of the proposed framework as applied to an instance of Minimum Vertex Cover. The middle part illustrates two iterations of the graph embedding, which results in node scores (green bars).

based on enumeration or branch-and-bound with an integer programming formulation, but may be prohibitive for large instances. On the other hand, polynomial-time approximation algorithms are desirable, but may suffer from weak optimality guarantees or empirical performance, or may not even exist for inapproximable problems. Heuristics are often fast, effective algorithms that lack theoretical guarantees, and may also require substantial problem-specific research and trial-and-error on the part of algorithm designers.

All three paradigms seldom exploit a common trait of real-world optimization problems: instances of the same type of problem are solved again and again on a regular basis, maintaining the same combinatorial structure, but differing mainly in their data. That is, in many applications, values of the coefficients in the objective function or constraints can be thought of as being sampled from the same underlying distribution. For instance, an advertiser on a social network targets a limited set of users with ads, in the hope that they spread them to their neighbors; such covering instances need to be solved repeatedly, since the influence pattern between neighbors may be different each time. Alternatively, a package delivery company routes trucks on a daily basis in a given city; thousands of similar optimizations need to be solved, since the underlying demand locations can differ.

Despite the inherent similarity between problem instances arising in the same domain, classical algorithms do not systematically exploit this fact. However, in industrial settings,

a company may be willing to invest in upfront, offline computation and learning if such a process can speed up its real-time decision-making and improve its quality. This motivates the main problem we address:

> **Problem Statement:** Given a graph optimization problem G and a distribution $\mathbb{D}$ of problem instances, can we learn better heuristics that generalize to unseen instances from $\mathbb{D}$?

Recently, there has been some seminal work on using deep architectures to learn heuristics for combinatorial problems, including the Traveling Salesman Problem [125, 19, 53]. However, the architectures used in these works are generic, not yet effectively reflecting the combinatorial structure of graph problems. As we show later, these architectures often require a huge number of instances in order to learn to generalize to new ones. Furthermore, existing works typically use the policy gradient for training [19], a method that is not particularly sample-efficient. While the methods in [125, 19] can be used on graphs with different sizes – a desirable trait – they require manual, ad-hoc input/output engineering to do so (e.g. padding with zeros).

In this paper, we address the challenge of learning algorithms for graph problems using a unique combination of reinforcement learning and graph embedding. The learned policy behaves like a meta-algorithm that incrementally constructs a solution, with the action being determined by a graph embedding network over the current state of the solution. More specifically, our proposed solution framework is different from previous work in the following aspects:

1. Algorithm design pattern. We will adopt a *greedy* meta-algorithm design, whereby a feasible solution is constructed by successive addition of nodes based on the graph structure, and is maintained so as to satisfy the problem's graph constraints. Greedy algorithms are a popular pattern for designing approximation and heuristic algorithms for graph problems. As such, the same high-level design can be seamlessly used for different graph optimization problems.

2. Algorithm representation. We will use a *graph embedding* network, `structure2vec` (S2V) [33], to represent the policy in the greedy algorithm. This novel deep learning architecture over the instance graph "featurizes" the nodes in the graph, capturing the properties of a node in the context of its graph neighborhood. This allows the policy to discriminate among nodes based on their usefulness, and generalizes to problem instances of different sizes. This contrasts with recent approaches [125, 19] that adopt a graph-agnostic sequence-to-sequence mapping that does not fully exploit graph structure.

3. Algorithm training. We will use fitted Q-learning to learn a greedy policy that is parametrized by the graph embedding network. The framework is set up in such a way that the policy will aim to optimize the objective function of the original problem instance *directly*. The main advantage of this approach is that it can deal with delayed rewards, which here represent the remaining increase in objective function value obtained by the greedy algorithm, in a data-efficient way; in each step of the greedy algorithm, the graph embeddings are updated according to the partial solution to reflect new knowledge of the benefit of *each node* to the final objective value. In contrast, the policy gradient approach of [19] updates the model parameters only once w.r.t. the whole solution (e.g. the tour in TSP).

The application of a greedy heuristic learned with our framework is illustrated in Figure 4.1. To demonstrate the effectiveness of the proposed framework, we apply it to three extensively studied graph optimization problems. Experimental results show that our framework, a single meta-learning algorithm, efficiently learns effective heuristics for each of the problems. Furthermore, we show that our learned heuristics preserve their effectiveness even when used on graphs much larger than the ones they were trained on. Since many combinatorial optimization problems, such as the set covering problem, can be explicitly or implicitly formulated on graphs, we believe that our work opens up a new avenue for graph algorithm design and discovery with deep learning.

4.2 Common Formulation for Greedy Algorithms on Graphs

We will illustrate our framework using three optimization problems over weighted graphs. Let $G(V, E, w)$ denote a weighted graph, where V is the set of nodes, E the set of edges and $w : E \to \mathbb{R}^+$ the edge weight function, i.e. $w(u, v)$ is the weight of edge $(u, v) \in E$. These problems are:

- **Minimum Vertex Cover (MVC):** Given a graph G, find a subset of nodes $S \subseteq V$ such that every edge is covered, i.e. $(u, v) \in E \Leftrightarrow u \in S$ or $v \in S$, and $|S|$ is minimized.
- **Maximum Cut (MAXCUT):** Given a graph G, find a subset of nodes $S \subseteq V$ such that the weight of the cut-set $\sum_{(u,v)\in C} w(u, v)$ is maximized, where cut-set $C \subseteq E$ is the set of edges with one end in S and the other end in $V \setminus S$.
- **Traveling Salesman Problem (TSP):** Given a set of points in 2-dimensional space, find a tour of minimum total weight, where the corresponding graph G has the points as nodes and is fully connected with edge weights corresponding to distances between points; a tour is a cycle that visits each node of the graph *exactly* once.

We will focus on a popular pattern for designing approximation and heuristic algorithms, namely a greedy algorithm. A greedy algorithm will construct a solution by sequentially adding nodes to a partial solution S, based on maximizing some *evaluation function* Q that measures the quality of a node in the context of the current partial solution. We will show that, despite the diversity of the combinatorial problems above, greedy algorithms for them can be expressed using a common formulation. Specifically:

1. A problem instance G of a given optimization problem is sampled from a distribution $\mathbb{D}$, i.e. the V, E and w of the instance graph G are generated according to a model or real-world data.
2. A partial solution is represented as an ordered list $S = (v_1, v_2, \ldots, v_{|S|})$, $v_i \in V$, and $\overline{S} = V \setminus S$ the set of candidate nodes for addition, conditional on S. Furthermore, we use a vector of binary decision variables x, with each dimension x_v corresponding to a

node $v \in V$, $x_v = 1$ if $v \in S$ and 0 otherwise. One can also view x_v as a tag or extra feature on v.

3. A maintenance (or helper) procedure $h(S)$ will be needed, which maps an ordered list S to a combinatorial structure satisfying the specific constraints of a problem.
4. The quality of a partial solution S is given by an objective function $c(h(S), G)$ based on the combinatorial structure h of S.
5. A generic greedy algorithm selects a node v to add next such that v maximizes an evaluation function, $Q(h(S), v) \in \mathbb{R}$, which depends on the combinatorial structure $h(S)$ of the current partial solution. Then, the partial solution S will be extended as

$$S \coloneqq (S, v^*), \quad \text{where } v^* \coloneqq \operatorname{argmax}_{v \in \overline{S}} Q(h(S), v), \tag{4.1}$$

and (S, v^*) denotes appending v^* to the end of a list S. This step is repeated until a termination criterion $t(h(S))$ is satisfied.

In our formulation, we assume that the distribution $\mathbb{D}$, the helper function h, the termination criterion t and the cost function c are all given. Given the above abstract model, various optimization problems can be expressed by using different helper functions, cost functions and termination criteria:

- **MVC:** The helper function does not need to do any work, and $c(h(S), G) = -|S|$. The termination criterion checks whether all edges have been covered.
- **MAXCUT:** The helper function divides V into two sets, S and its complement $\overline{S} = V \setminus S$, and maintains a cut-set $C = \{(u, v) \,|\, (u, v) \in E, u \in S, v \in \overline{S}\}$. Then, the cost is $c(h(S), G) = \sum_{(u,v) \in C} w(u, v)$, and the termination criterion does nothing.
- **TSP:** The helper function will maintain a tour according to the order of the nodes in S. The simplest way is to append nodes to the end of partial tour in the same order as S. Then the cost $c(h(S), G) = -\sum_{i=1}^{|S|-1} w(S(i), S(i+1)) - w(S(|S|), S(1))$, and the termination criterion is activated when $S = V$. Empirically, inserting a node u in the partial tour at the position which increases the tour length the least is a better choice. We adopt this insertion procedure as a helper function for TSP.

An estimate of the quality of the solution resulting from adding a node to partial solution S will be determined by the *evaluation function* Q, which will be learned using a collection of problem instances. This is in contrast with traditional greedy algorithm design, where the *evaluation function* Q is typically hand-crafted, and requires substantial problem-specific research and trial-and-error. In the following, we will design a powerful deep learning parameterization for the evaluation function, $\hat{Q}(h(S), v; \Theta)$, with parameters Θ.

4.3 Representation: Graph Embedding

Since we are optimizing over a graph G, we expect that the evaluation function $\hat{Q}$ should take into account the current partial solution S as it maps to the graph. That is, $x_v = 1$ for all nodes $v \in S$, and the nodes are connected according to the graph structure. Intuitively, $\hat{Q}$ should summarize the state of such a "tagged" graph G, and figure out the value of a new node if it is to be added in the context of such a graph. Here, both the state of the graph and the context of a node v can be very complex, hard to describe in closed form, and may depend on complicated statistics such as global/local degree distribution, triangle counts, distance to tagged nodes, etc. In order to represent such complex phenomena over combinatorial structures, we will leverage a deep learning architecture over graphs, in particular the `structure2vec` of [33], to parameterize $\hat{Q}(h(S), v; \Theta)$.

4.3.1 Structure2Vec

We first provide an introduction to `structure2vec`. This graph embedding network will compute a p-dimensional feature embedding μ_v for each node $v \in V$, given the current partial solution S. More specifically, `structure2vec` defines the network architecture recursively according to an input graph structure G, and the computation graph of `structure2vec` is inspired by graphical model inference algorithms, where node-specific tags or features x_v are aggregated recursively according to G's graph topology. After a few steps of recursion, the network will produce a new embedding for each node, taking

into account both graph characteristics and long-range interactions between these node features. One variant of the `structure2vec` architecture will initialize the embedding $\mu_v^{(0)}$ at each node as 0, and for all $v \in V$ update the embeddings synchronously at each iteration as

$$\mu_v^{(t+1)} \leftarrow F\left(x_v, \{\mu_u^{(t)}\}_{u\in\mathcal{N}(v)}, \{w(v,u)\}_{u\in\mathcal{N}(v)}; \Theta\right), \tag{4.2}$$

where $\mathcal{N}(v)$ is the set of neighbors of node v in graph G, and F is a generic nonlinear mapping such as a neural network or kernel function.

Based on the update formula, one can see that the embedding update process is carried out based on the graph topology. A new round of embedding sweeping across the nodes will start only after the embedding update for all nodes from the previous round has finished. It is easy to see that the update also defines a process where the node features x_v are propagated to other nodes via the nonlinear propagation function F. Furthermore, the more update iterations one carries out, the farther away the node features will propagate and get aggregated nonlinearly at distant nodes. In the end, if one terminates after T iterations, each node embedding $\mu_v^{(T)}$ will contain information about its T-hop neighborhood as determined by graph topology, the involved node features and the propagation function F. An illustration of two iterations of graph embedding can be found in Figure 4.1.

4.3.2 Parameterizing $\hat{Q}(h(S), v; \Theta)$

We now discuss the parameterization of $\hat{Q}(h(S), v; \Theta)$ using the embeddings from `structure2vec`. In particular, we design F to update a p-dimensional embedding μ_v as:

$$\mu_v^{(t+1)} \leftarrow \mathrm{relu}\big(\theta_1 x_v + \theta_2 \sum\nolimits_{u\in\mathcal{N}(v)} \mu_u^{(t)} + \theta_3 \sum\nolimits_{u\in\mathcal{N}(v)} \mathrm{relu}(\theta_4\, w(v,u))\big), \tag{4.3}$$

where $\theta_1 \in \mathbb{R}^p$, $\theta_2, \theta_3 \in \mathbb{R}^{p\times p}$ and $\theta_4 \in \mathbb{R}^p$ are the model parameters, and relu is the rectified linear unit ($\mathrm{relu}(z) = \max(0, z)$) applied elementwise to its input. The summation over neighbors is one way of aggregating neighborhood information that is invariant to permutations over neighbors. For simplicity of exposition, x_v here is a binary scalar as

described earlier; it is straightforward to extend x_v to a vector representation by incorporating any additional useful node information. To make the nonlinear transformations more powerful, we can add some more layers of relu before we pool over the neighboring embeddings μ_u.

Once the embedding for each node is computed after T iterations, we will use these embeddings to define the $\hat{Q}(h(S), v; \Theta)$ function. More specifically, we will use the embedding $\mu_v^{(T)}$ for node v and the pooled embedding over the entire graph, $\sum_{u \in V} \mu_u^{(T)}$, as the surrogates for v and $h(S)$, respectively, i.e.

$$\hat{Q}(h(S), v; \Theta) = \theta_5^\top \, \mathrm{relu}([\theta_6 \sum\nolimits_{u \in V} \mu_u^{(T)}, \theta_7 \, \mu_v^{(T)}]) \tag{4.4}$$

where $\theta_5 \in \mathbb{R}^{2p}$, $\theta_6, \theta_7 \in \mathbb{R}^{p \times p}$ and $[\cdot, \cdot]$ is the concatenation operator. Since the embedding $\mu_u^{(T)}$ is computed based on the parameters from the graph embedding network, $\hat{Q}(h(S), v)$ will depend on a collection of 7 parameters $\Theta = \{\theta_i\}_{i=1}^7$. The number of iterations T for the graph embedding computation is usually small, such as $T = 4$.

The parameters Θ will be learned. Previously, [33] required a ground truth label for every input graph G in order to train the `structure2vec` architecture. There, the output of the embedding is linked with a softmax-layer, so that the parameters can by trained end-to-end by minimizing the cross-entropy loss. This approach is not applicable to our case due to the lack of training labels. Instead, we train these parameters together *end-to-end* using reinforcement learning.

4.4 Training: Q-learning

We show how reinforcement learning is a natural framework for learning the evaluation function $\hat{Q}$. The definition of the evaluation function $\hat{Q}$ naturally lends itself to a *reinforcement learning* (RL) formulation [120], and we will use $\hat{Q}$ as a model for the state-value function in RL. We note that we would like to learn a function $\hat{Q}$ *across a set of* m *graphs from distribution* $\mathbb{D}$, $\mathcal{D} = \{G_i\}_{i=1}^m$, with potentially different sizes. The advantage of the

graph embedding parameterization in our previous section is that we can deal with different graph instances and sizes seamlessly.

4.4.1 Reinforcement learning formulation

We define the states, actions and rewards in the reinforcement learning framework as follows:

1. *States*: a state S is a sequence of actions (nodes) on a graph G. Since we have already represented nodes in the tagged graph with their embeddings, the state is a vector in p-dimensional space, $\sum_{v \in V} \mu_v$. It is easy to see that this embedding representation of the state can be used across different graphs. The terminal state $\hat{S}$ will depend on the problem at hand;
2. *Transition*: transition is deterministic here, and corresponds to tagging the node $v \in G$ that was selected as the last action with feature $x_v = 1$;
3. *Actions*: an action v is a node of G that is not part of the current state S. Similarly, we will represent actions as their corresponding p-dimensional node embedding μ_v, and such a definition is applicable across graphs of various sizes;
4. *Rewards*: the reward function $r(S, v)$ at state S is defined as the change in the cost function after taking action v and transitioning to a new state $S' \coloneqq (S, v)$. That is,

$$r(S, v) = c(h(S'), G) - c(h(S), G), \tag{4.5}$$

   and $c(h(\emptyset), G) = 0$. As such, the *cumulative reward* R of a terminal state $\hat{S}$ coincides exactly with the objective function value of the $\hat{S}$, i.e. $R(\hat{S}) = \sum_{i=1}^{|\hat{S}|} r(S_i, v_i)$ is equal to $c(h(\hat{S}), G)$;
5. *Policy*: based on $\hat{Q}$, a deterministic greedy policy $\pi(v|S) \coloneqq \operatorname{argmax}_{v' \in \overline{S}} \hat{Q}(h(S), v')$ will be used. Selecting action v corresponds to adding a node of G to the current partial solution, which results in collecting a reward $r(S, v)$.

Table 4.1 shows the instantiations of the reinforcement learning framework for the three optimization problems considered herein. We let Q^* denote the *optimal* Q-function for each RL problem. Our graph embedding parameterization $\hat{Q}(h(S), v; \Theta)$ from Section 4.3 will

Table 4.1: Definition of reinforcement learning components for each of the three problems considered.

Problem	State	Action	Helper function	Reward	Termination
MVC	subset of nodes selected so far	add node to subset	None	-1	all edges are covered
MAXCUT	subset of nodes selected so far	add node to subset	None	change in cut weight	cut weight cannot be improved
TSP	partial tour	grow tour by one node	Insertion operation	change in tour cost	tour includes all nodes

then be a function approximation model for it, which will be learned via n-step Q-learning.

4.4.2 Learning algorithm

In order to perform end-to-end learning of the parameters in $\hat{Q}(h(S), v; \Theta)$, we use a combination of n-step Q-learning [120] and *fitted Q-iteration* [111], as illustrated in Algorithm 1. We use the term *episode* to refer to a complete sequence of node additions starting from an empty solution, and until termination; a *step* within an episode is a single action (node addition).

Standard (1-step) Q-learning updates the function approximator's parameters *at each step* of an episode by performing a gradient step to minimize the squared loss:

$$(y - \hat{Q}(h(S_t), v_t; \Theta))^2, \tag{4.6}$$

where $y = \gamma \max_{v'} \hat{Q}(h(S_{t+1}), v'; \Theta) + r(S_t, v_t)$ for a non-terminal state S_t. The n-step Q-learning helps deal with the issue of *delayed rewards*, where the final reward of interest to the agent is only received far in the future during an episode. In our setting, the final objective value of a solution is only revealed after many node additions. As such, the 1-step update may be too myopic. A natural extension of 1-step Q-learning is to wait n steps before updating the approximator's parameters, so as to collect a more accurate estimate of the future rewards. Formally, the update is over the same squared loss (4.6), but with a different target, $y = \sum_{i=0}^{n-1} r(S_{t+i}, v_{t+i}) + \gamma \max_{v'} \hat{Q}(h(S_{t+n}), v'; \Theta)$. The fitted Q-iteration approach has been shown to result in faster learning convergence when using a neural network as a function approximator [111, 96], a property that also applies in our setting, as we use the embedding defined in Section 4.3.2. Instead of updating the Q-function sample-by-sample as in Equation (4.6), the fitted Q-iteration approach uses *experience*

replay to update the function approximator with a batch of samples from a dataset E, rather than the single sample being currently experienced. The dataset E is populated during previous episodes, such that at step $t+n$, the tuple $(S_t, a_t, R_{t,t+n}, S_{t+n})$ is added to E, with $R_{t,t+n} = \sum_{i=0}^{n-1} r(S_{t+i}, a_{t+i})$. Instead of performing a gradient step in the loss of the current sample as in (4.6), stochastic gradient descent updates are performed on a random sample of tuples drawn from E.

It is known that *off-policy* reinforcement learning algorithms such as Q-learning can be more sample efficient than their policy gradient counterparts [54]. This is largely due to the fact that policy gradient methods require *on-policy* samples for the new policy obtained after each parameter update of the function approximator.

Algorithm 1 Q-learning for the Greedy Algorithm

Initialize experience replay memory $\mathcal{M}$ to capacity N
for episode $e = 1$ **to** L **do**
  Draw graph G from distribution $\mathbb{D}$
  Initialize the state to empty $S_1 = ()$
  for step $t = 1$ **to** T **do**
    $v_t = \begin{cases} \text{random node } v \in \overline{S}_t, & \text{w.p. } \epsilon \\ \text{argmax}_{v \in \overline{S}_t} \hat{Q}(h(S_t), v; \Theta), & \text{otherwise} \end{cases}$
    Add v_t to partial solution: $S_{t+1} \coloneqq (S_t, v_t)$
    if $t \geq n$ **then**
      Add tuple $(S_{t-n}, v_{t-n}, R_{t-n,t}, S_t)$ to $\mathcal{M}$
      Sample random batch from $B \overset{iid.}{\sim} \mathcal{M}$
      Update Θ by SGD over (4.6) for B
    end if
  end for
end for
return Θ

4.5 Experimental Evaluation

Instance generation. To evaluate the proposed method against other approximation/heuristic algorithms and deep learning approaches, we generate graph instances for each of the three problems. For the MVC and MAXCUT problems, we generate Erdős-Renyi (ER) [42] and Barabasi-Albert (BA) [10] graphs which have been used to model many real-world networks. For a given range on the number of nodes, e.g. 50-100, we first sample the number of nodes uniformly at random from that range, then generate a graph according to either ER or BA. For the two-dimensional TSP problem, we use an instance generator from the DIMACS TSP Challenge [69] to generate uniformly random or clustered points in the 2-D grid. We refer the reader to the Appendix A.3.1 for complete details on instance generation. We have also tackled the Set Covering Problem, for which the description and results are deferred to Appendix A.1.

Structure2Vec Deep Q-learning. For our method, S2V-DQN, we use the graph representations and hyperparameters described in Appendix A.3.4. The hyperparameters are selected via preliminary results on small graphs, and then fixed for large ones. Note that for TSP, where the graph is fully-connected, we build the K-nearest neighbor graph ($K = 10$) to scale up to large graphs. For MVC, where we train the model on graphs with up to 500 nodes, we use the model trained on small graphs as initialization for training on larger ones. We refer to this trick as “pre-training", which is illustrated in Figure A.2.

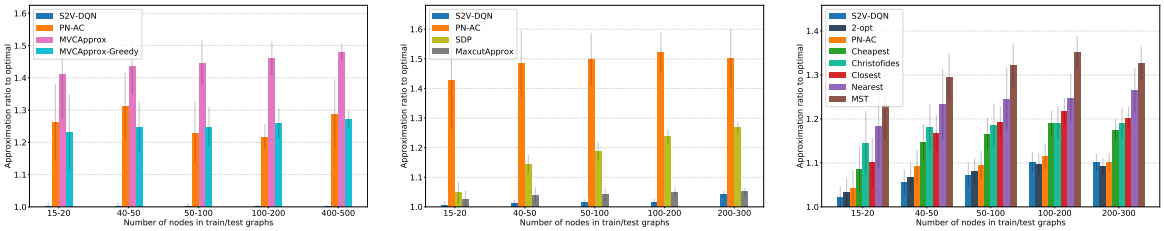
Pointer Networks with Actor-Critic. We compare our method to a method, based on Recurrent Neural Networks (RNNs), which does not make full use of graph structure [19]. We implement and train their algorithm (PN-AC) for all three problems. The original model only works on the Euclidian TSP problem, where each node is represented by its (x, y) coordinates, and is not designed for problems with graph structure. To handle other graph problems, we describe each node by its adjacency vector instead of coordinates. To handle different graph sizes, we use a singular value decomposition (SVD) to obtain a rank-8

approximation for the adjacency matrix, and use the low-rank embeddings as inputs to the pointer network.

Baseline Algorithms. Besides the PN-AC, we also include powerful approximation or heuristic algorithms from the literature. These algorithms are specifically designed for each type of problem:

- **MVC:** *MVCApprox* iteratively selects an uncovered edge and adds both of its endpoints [103]. We designed a stronger variant, called *MVCApprox-Greedy*, that greedily picks the uncovered edge with maximum sum of degrees of its endpoints. Both algorithms are 2-approximations.
- **MAXCUT:** We include *MaxcutApprox*, which maintains the cut set $(S, V \setminus S)$ and moves a node from one side to the other side of the cut if that operation results in cut weight improvement [80]. To make *MaxcutApprox* stronger, we greedily move the node that results in the largest improvement in cut weight. A randomized, non-greedy algorithm, referred to as SDP, is also implemented based on [50]; 100 solutions are generated for each graph, and the best one is taken.
- **TSP:** We include the following approximation algorithms: Minimum Spanning Tree (MST), Cheapest insertion (Cheapest), Closest insertion (Closest), Christofides and 2-opt. We also add the Nearest Neighbor heuristic (Nearest); see [16] for algorithmic details.

Details on Validation and Testing. For S2V-DQN and PN-AC, we use a CUDA K80-enabled cluster for training and testing. Training convergence for S2V-DQN is discussed in Appendix A.3.6. S2V-DQN and PN-AC use 100 held-out graphs for validation, and we report the test results on another 1000 graphs. We use CPLEX[65] to get optimal solutions for MVC and MAXCUT, and Concorde [15] for TSP (details in Appendix A.3.1). All approximation ratios reported in the paper are with respect to the best (possibly optimal) solution found by the solvers within 1 hour. For MVC, we vary the training and test graph sizes in the ranges $\{$15–20, 40–50, 50–100, 100–200, 400–500$\}$. For MAXCUT and TSP, which involve edge weights, we train up to 200–300 nodes due to the limited computation



(a) MVC BA (b) MAXCUT BA (c) TSP clustered

Figure 4.2: Approximation ratio on 1000 test graphs. Note that on MVC, our performance is pretty close to optimal. In this figure, training and testing graphs are generated according to the same distribution.

resource. For all problems, we test on graphs of size up to 1000–1200.

During testing, instead of using Active Search as in [19], we simply use the greedy policy. This gives us much faster inference, while still being powerful enough. We modify existing open-source code to implement both S2V-DQN [1] and PN-AC [2]. Our code is publicly available [3].

4.5.1 Comparison of solution quality

To evaluate the solution quality on test instances, we use the *approximation ratio* of each method relative to the optimal solution, averaged over the set of test instances. The approximation ratio of a solution S to a problem instance G is defined as $\mathcal{R}(S,G) = \max(\frac{OPT(G)}{c(h(S))}, \frac{c(h(S))}{OPT(G)})$, where $c(h(S))$ is the objective value of solution S, and $OPT(G)$ is the best-known solution value of instance G.

Figure 4.2 shows the average approximation ratio across the three problems; other graph types are in Figure A.1 in the appendix. In all of these figures, a lower approximation ratio is better. Overall, our proposed method, S2V-DQN, performs significantly better than other methods. In MVC, the performance of S2V-DQN is particularly good, as the approximation ratio is roughly 1 and the bar is barely visible.

The PN-AC algorithm performs well on TSP, as expected. Since the TSP graph is essentially fully-connected, graph structure is not as important. On problems such as

[1] https://github.com/Hanjun-Dai/graphnn
[2] https://github.com/devsisters/pointer-network-tensorflow
[3] https://github.com/Hanjun-Dai/graph_comb_opt

MVC and MAXCUT, where graph information is more crucial, our algorithm performs significantly better than PN-AC. For TSP, The 2-opt algorithm performs as well as S2V-DQN, and slightly better in some cases, an intuitive result given the sophistication of this algorithm, which exchanges pairs of edges that can give a smaller tour.

4.5.2 Generalization to larger instances

The graph embedding framework enables us to train and test on graphs of different sizes, since the same set of model parameters are used. How does the performance of the learned algorithm using small graphs generalize to test graphs of larger sizes? To investigate this, we train S2V-DQN on graphs with 50–100 nodes, and test its generalization ability on graphs with up to 1200 nodes. Table 4.2 summarizes the results, and full results are in Appendix A.3.3.

Table 4.2: S2V-DQN's generalization ability. Values are average approximation ratios over 1000 test instances. These test results are produced by S2V-DQN algorithms trained on graphs with 50-100 nodes.

Test Size	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
MVC (BA)	1.0033	1.0041	1.0045	1.0040	1.0045	1.0048	1.0062
MAXCUT (BA)	1.0150	1.0181	1.0202	1.0188	1.0123	1.0177	1.0038
TSP (clustered)	1.0730	1.0895	1.0869	1.0918	1.0944	1.0975	1.1065

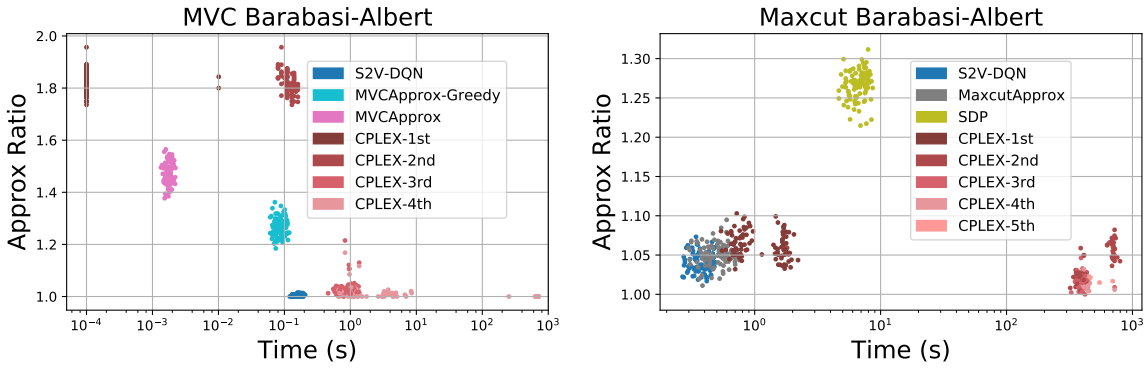
We can see that S2V-DQN achieves a very good approximation ratio. Note that the "optimal" value used in the computation of approximation ratios may not be truly optimal (due to the solver time cutoff at 1 hour), and so CPLEX's solutions do typically get worse as problem size grows. This is why sometimes we can even get better approximation ratio on larger graphs.

4.5.3 Scalability & The Time-Quality Trade-off

To construct a solution on a test graph, our algorithm has polynomial complexity of $O(k|E|)$ where k is number of greedy steps (at most the number of nodes $|V|$) and $|E|$ is number of

edges. For instance, on graphs with 1200 nodes, we can find the solution of MVC within 11 seconds using a single GPU, while getting an approximation ratio of 1.0062. For dense graphs, we can also sample the edges for the graph embedding computation to save time, a measure we will investigate in the future.

Figure 4.3 illustrates the approximation ratios of various approaches as a function of running time. All algorithms report a single solution at termination, whereas CPLEX reports multiple improving solutions, for which we recorded the corresponding running time and approximation ratio. Figure A.3 (Appendix A.3.7) includes other graph sizes and types, where the results are consistent with Figure 4.3.



(a) MVC BA 200-300 (b) MAXCUT BA 200-300

Figure 4.3: Time-approximation trade-off for MVC and MAXCUT. In this figure, each dot represents a solution found for a single problem instance, for 100 instances. For CPLEX, we also record the time and quality of each solution it finds, e.g. CPLEX-1st means the first feasible solution found by CPLEX.

Figure 4.3 shows that, for MVC, we are slightly slower than the approximation algorithms but enjoy a much better approximation ratio. Also note that although CPLEX found the first feasible solution quickly, it also has much worse ratio; the second improved solution found by CPLEX takes similar or longer time than our S2V-DQN, but is still of worse quality. For MAXCUT, the observations are still consistent. One should be aware that sometimes our algorithm can obtain better results than 1-hour CPLEX, which gives ratios below 1.0. Furthermore, sometimes S2V-DQN is even faster than the *MaxcutApprox*, although this comparison is not exactly fair, since we use GPUs; however, we can still see

that our algorithm is efficient.

4.5.4 Experiments on real-world datasets

In addition to the experiments for synthetic data, we identified sets of publicly available benchmark or real-world instances for each problem, and performed experiments on them. A summary of results is in Table 4.3, and details are given in Appendix A.2. S2V-DQN significantly outperforms all competing methods for MVC, MAXCUT and TSP.

Table 4.3: Realistic data experiments, results summary. Values are average approximation ratios.

Problem	**Dataset**	**S2V-DQN**	**Best Competitor**	**2nd Best Competitor**
MVC	MemeTracker	**1.0021**	1.2220 (MVCApprox-Greedy)	1.4080 (MVCApprox)
MAXCUT	Physics	**1.0223**	1.2825 (MaxcutApprox)	1.8996 (SDP)
TSP	TSPLIB	**1.0475**	1.0947 (2-opt)	1.1771 (Cheapest)

4.5.5 Discovery of interesting new algorithms

We further examined the algorithms learned by S2V-DQN, and tried to interpret what greedy heuristics have been learned. We found that S2V-DQN is able to discover new and interesting algorithms which intuitively make sense but have not been analyzed before. For instance, S2V-DQN discovers an algorithm for MVC where nodes are selected to balance between their degrees and the connectivity of the remaining graph (Appendix Figures A.4 and A.7). For MAXCUT, S2V-DQN discovers an algorithm where nodes are picked to avoid cancelling out existing edges in the cut set (Appendix Figure A.5). These results suggest that S2V-DQN may also be a good assistive tool for discovering new algorithms, especially in cases when the graph optimization problems are new and less well-studied.

4.6 Conclusions

We presented an end-to-end machine learning framework for automatically designing greedy heuristics for hard combinatorial optimization problems on graphs. Central to our approach

is the combination of a deep graph embedding with reinforcement learning. Through extensive experimental evaluation, we demonstrate the effectiveness of the proposed framework in learning greedy heuristics as compared to manually-designed greedy algorithms. The excellent performance of the learned heuristics is consistent across multiple different problems, graph types, and graph sizes, suggesting that the framework is a promising new tool for designing algorithms for graph problems.

CHAPTER 5
NEURAL INTEGER OPTIMIZATION

The design of algorithms for hard discrete optimization problems often follows one of two paradigms: general algorithms, heuristic or exact, that operate on a wide range of problems, e.g. branch-and-bound or local search; or specialized heuristics that exploit the structure of a given problem. Both paradigms rely on the ingenuity of the algorithm designer, coupled with trial-and-error on a set of instances of interest. Recently, there has been a surge in research on learning heuristics for combinatorial optimization problems over a distribution of instances, a departure from the classical paradigms mentioned earlier. Despite promising results on problems such as the TSP, existing learning methods are only capable of handling simple constraints, e.g. subtour constraints for insertion heuristics in TSP. Our work is an attempt at learning heuristics for discrete optimization problems subject to generic constraints, i.e. constraints that may be hard to satisfy, e.g. general linear inequalities. The key contribution of this work lies in incorporating projection into a recurrent neural network model that generates solutions to a discrete optimization problem with intricate constraints. We apply our framework to 0-1 linear optimization problems, and show promising results on instances from Two-Stage Stochastic Programming, the Generalized Assignment Problem and the Satisfiability problem.

5.1 Introduction

NP-hard combinatorial optimization problems are at the heart of many complex decision-making tasks. In particular, *mixed-integer linear programs* (MIP), involving a mixture of discrete and continuous variables as well as linear constraints and linear objective, is a very broad class of discrete optimization optimization problems. Formally, a MIP instance is

described as follows:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & c^T x \\ \text{subject to} \quad & Ax \leq b, \\ & l \leq x \leq u, \\ & x_j \in \{0,1\}\ \forall j \in \mathcal{B}. \end{aligned} \tag{5.1}$$

Mixed Integer Programming (MIP) has been used in diverse domains such as aircraft routing, wildlife conservation, sports scheduling, dose distribution and kidney exchange,data center optimization, and data mining tasks, among many more. Hence, improving our ability to solve MIP problems effectively can be a significant contribution across various domains.

The Branch-and-Bound algorithm is typically used to find optimal MIP solutions and certify their optimality. However, in many practical situation, one might be willing to settle for good, but not necessarily optimal solutions to their NP-Hard problem. While many specialized heuristics have been developed for specific families of problems (e.g. knapsack, set covering, independent set, etc.), there has also been significant interest in designing *general-purpose heuristics* for binary programming. A general-purpose heuristic operates solely on a MIP instance (5.1) without any prior knowledge of the combinatorial structure of the problem. Such heuristics are valuable for two reasons:

– They can be used “out-of-the-box" on a new unexplored optimization problem, for which no specialized heuristics have been developed yet;

– They can be directly incorporated into exact MIP solvers, providing feasible solutions during branch-and-bound search that can help prune suboptimal regions in the search space.

The “Feasibility Pump" (FP) [43] is arguably the most widely used general-purpose heuristic for MIP; see [23] for a recent survey on the topic. However, heuristics such as the FP are designed with the aim of obtaining feasible solutions on any arbitrary MIP

instance, without regard to its structure. This may be seen as a weakness when compared to specialized heuristics, which when available are able to take considerable advantage of structure.

In this work, we take serious steps towards bridging the gap between general-purpose heuristics and specialized heuristics for integer programming. To do so, we *propose a data-driven learning-based approach to automatically designing an effective general-purpose MIP heuristic*, over a distribution of instances. We show that our approach is able to learn a model that drives a heuristic for finding feasible solutions for any MIP (without any restriction on the type of linear constraints it involves), and that the learned heuristics are able to generalize to larger instances of the same problem domain as well as across domains.

5.2 Related Work

There has been a rising interest and success in improving optimization algorithms by integrating machine learning within solvers. Some such examples include using deep learning to tune gradient descent [14], reinforcement learning used for job-shop scheduling [128], and classification used for selecting an algorithm for QBF subproblems [114]. Closely related to our work is [14], where a recurrent neural network model is used for continuous optimization. In comparison, the main novelty in our work lies in successfully tackling *discrete optimization* problems with linear constraints, both missing aspects in [14].

Perhaps most related to our work is the recently proposed NeuroSAT learning framework for satisfiability [116]. NeuroSAT uses graph neural networks to embed the decision variables and constraints, and attempts to predict a feasible solution based on the embeddings. In contrast, we do not use any graph embeddings and adopt a simple feature representation of variables. This makes our models much more sample-efficient, resulting in better performance on SAT instances used in [116]; Section 5.5.3 details this empirical comparison.

In the MIP literature, recently a number of tasks within Branch-and-Bound have successfully leveraged ML, including: branching [77, 12], node selection [56], decompositions [83],

formulation selection [27], parameter configuration [127]. These works are orthogonal to our paper, in that they tackle exact solving whereas we are interested in learning effective heuristics to integer programs.

5.3 Neural Integer Optimization

Given that binary variables are predominant in real-world MIPs, we restrict our attention to MIP problems with pure binary variables; continuous variables can be trivially accommodated in the methods we developed. A MIP with non-binary integer variables can be easily transformed into an equivalent binary MIP.

We refer to the values in A as "constraint coefficients". The Linear Programming (LP) relaxation of the MIP removes the 0-1 integrality constraint and allows the variables in $\mathcal{B}$ to take on continuous values in $[0, 1]$. The LP-feasible region is then defined as:

$$P \coloneqq \{x \in \mathbb{R}^n \mid Ax \leq b, l \leq x \leq u\}.$$

5.3.1 Problem Statement

We are given a dataset $\mathcal{I} \coloneqq \{(A_i, b_i, c_i, \mathcal{B}_i)_{i=1}^d\}$ consisting of d MIP instances, as defined by their constraint matrices A_i, right hand-side vectors b_i, objective coefficients c_i and sets of binary variables $\mathcal{B}_i$. The MIP instances in $\mathcal{I}$ are assumed to belong to the same distribution of problems, broadly defined, though they may have a different number of variables and constraints. Section 5.4 illustrates such instance distributions in the context of some practical optimization problems.

We define an algorithm $\mathcal{A}$, parametrized by a vector $\Theta \in \mathbb{R}^p$, as a mapping from a MIP instance I to $\{0, 1\}$, where 1 corresponds to the event "algorithm $\mathcal{A}$ finds a solution to instance I", and 0 corresponds to not finding a solution.

Given a dataset $\mathcal{I}$ of MIP instances as defined above, the learning task then becomes that of finding a parameter setting for algorithm $\mathcal{A}$ such that we maximize the number of

instances for which $\mathcal{A}$ finds a feasible solution:

$$\Theta^* := \arg\max_{\Theta \in \mathbb{R}^p} \frac{1}{d} \sum_{I \in \mathcal{I}} \mathcal{A}(I; \Theta) \tag{5.2}$$

Clearly, solving this optimization problem is only useful if the resulting learned algorithm can perform well out-of-sample, i.e. on unseen MIP instances from the same distribution of problems; most of our Experiments section is dedicated to empirically answering that question.

Towards solving problem (5.2), we pose and tackle the following three questions:

1. **What kind of algorithm is $\mathcal{A}$?** For example, is it a greedy or local search algorithm that repeatedly flips binary variables until a feasible solution is found? Section 5.3.2 addresses this issue.

2. **What are we learning exactly?** How does the choice of parameters Θ impact algorithm $\mathcal{A}$'s behavior? Section 5.3.3 will present a key idea that resolves this question: whatever is learned should explicitly take the constraints $Ax \leq b$ into account.

3. **How can we solve the optimization problem** (5.2)**?** More specifically, is this a supervised or reinforcement learning problem? We show in Section 5.3.4 that it can simply be cast as a function optimization problem.

5.3.2 Algorithmic Framework

In the spirit of the Feasibility Pump [43], $\mathcal{A}$ is an *iterative projection* algorithm. For an instance $I = (A, b, c, \mathcal{B})$ with LP-feasible region P, $\mathcal{A}$ initially receives x_0, an optimal solution to the LP relaxation of I. $\mathcal{A}$ then transforms x_0 into $x_1 \in P$, another LP-feasible point. The key idea is that as it moves from x^t to x^{t+1}, the algorithm tries to increase the number of integer variables in x^{t+1}, as compared to x^t. Whenever $x^{t'}$ becomes integer-feasible in addition to being LP-feasible, then a feasible solution has been found and the algorithm terminates.

Why is this algorithmic framework a reasonable choice? There are certain MIP problems that admit much simpler strategies such as greedy construction. For instance, a greedy selection policy has been learned for some graph optimization problems. We argue that such problems have *highly structured* constraints that make it easy or trivial to find feasible solutions. Consider the Minimum Vertex Cover (VC) problem, for example, where one seeks a minimal subset of nodes in a graph such that every edge is covered by at least one node in the chosen subset. A greedy algorithm for VC, learned or otherwise, can simply start from an empty set and grow it until all edges are covered.

If the structure of the constraints is intricate, heavier machinery must be invoked to maintain feasibility in the constraints until an integer solution is found. The “transformation" mentioned earlier is simply a *projection* in the form of the following LP:

$$x^{t+1} = \arg\min_{x \in P} p^T x. \tag{5.3}$$

Clearly, the key to a successful algorithm is the choice of the projection coefficients vector p. This is where the learnable parameters Θ come into play.

5.3.3 Learning to Project

We adopt a simple Recurrent Neural Network (RNN) model, analogous to that of [14], that processes each variable separately (“coordinate-wise"), and maintains a per-variable hidden state. The actual neural network module we use is a Gated Recurrent Unit (GRU). The GRU takes as input two simple features of each variable: the value of the variable j from the previous iteration, x_j^t, and its rounding to the nearest integer, thereafter referred to as $[x_j^t]$. The GRU is followed by a learnable linear layer that transforms the final hidden state of a variable into a projection coefficient p_j^t. The projection coefficients of all variables are then fed into the LP projection (5.3).

5.3.4 Learning Procedure

In contrast with recent approaches for learning in combinatorial optimization, the proposed framework does not require neither supervision (in the form of optimal solutions to training instances as in [125]), nor reinforcement learning as in [34, 19]. Instead, we view Problem 5.2 on a dataset of training instances $\mathcal{I}$ as a function optimization problem in which the target values are provided "for free". Specifically, at a given iteration of algorithm $\mathcal{A}$, we consider that a variable x_j with value $x^t > 0.5$ should be pushed towards the nearest integer, i.e. 1. The Binary Cross-Entropy (BCE) can be used as loss function in our setting on a given instance I as follows:

$$BCE(I;\Theta) = -\sum_{t=1}^{T}\sum_{j\in\mathcal{B}} [x_j^t]\cdot\log x_j^t + (1-[x_j^t])\cdot\log(1-x_j^t), \tag{5.4}$$

where T is the total number of iterations of the algorithm. The final BCE loss function that we minimize as a surrogate to our true objective is just the sum of (4) over all instances.

Table 5.1: Statistics on the datasets used. We only train/validate on the smallest sets of instances and test on instances of the same and larger sizes.

	Problem Parameters	**# Variables**	**# Constraints**	**Training**	**Validation**	**Testing**
	(3, 20)	60	23	✗	✗	✗
GAP	(5, 50)	150	55			✗
(processors, tasks)	(5, 100)	500	105			✗
	(10, 10)	110	100	✗	✗	✗
STOC	(10, 20)	220	200			✗
(k, p)	(20, 10)	210	200			✗

5.4 Optimization Problems

This section introduces the optimization problems that we will use for experimental evaluation in Section 6.5. Table 5.1 provides details on the problem sets generated and their sizes. We chose these problems for the following reasons: (1) they are widely used to model real-world decision making tasks, and (2) have intricate constraint structure that makes finding feasible solutions non-trivial.

Generalized Assignment Problem

In the *Generalized Assignment Problem* (GAP), we are given a set of n tasks and m processors. A task j incurs a cost of $w_{ij} \in \mathbb{R}_{\geq 0}$ and produces a profit of $p_{ij} \in \mathbb{R}_{\geq 0}$ if assigned to processor i. Processor i has a cost capacity of $c_i \in \mathbb{R}_{>0}$. The optimization problem is then to assign each task to exactly one processor such that the processor capacities are not exceeded and the total profit is maximized. GAP finds applications in diverse domains such as computer systems, e.g. in data storage and distributed caching [36, 46], telecommunications and routing [29], as well as operations research, e.g. production planning [39].

The mathematical formulation of GAP is as follows:

$$\begin{aligned}
\underset{x}{\text{maximize}} \quad & \sum_{i=1}^{m} \sum_{j=1}^{n} p_{ij} x_{ij} \\
\text{subject to} \quad & \sum_{j=1}^{n} w_{ij} x_{ij} \leq c_i, \ i = 1, \ldots, m, \\
& \sum_{i=1}^{m} x_{ij} = 1, \ j = 1, \ldots, n, \\
& x_{ij} \in \{0, 1\}, \ i = 1, \ldots, m, j = 1, \ldots, n.
\end{aligned}$$

GAP has $n \cdot m$ variables and $n + m$ constraints. We generate instances of GAP according to the distribution "(c)" proposed in [93]: For a given n and m, w_{ij} and p_{ij} are drawn uniformly at random from $[5, 25]$ and $[1, 40]$, respectively. The capacities are set as: $c_i = 0.8 \sum_{j=1}^{n} w_{ij}/m$. Table 5.1provides additional details on the GAP instances we generated for training, validation and testing.

Two-Stage Stochastic Programming

Consider the following inventory management problem [37]: a retailer has to order a certain amount of each product in order to satisfy demand. The retailer must decide on the amounts to order before the demand becomes known to them. Once the demand is revealed, they can order additional product if the demand has not been met by the first-stage orders. Assuming

the demand is random, the problem we just described is a *two-stage stochastic programming* problem [8]. A typical approach to solving such problems consists in selecting a set of highly probable "scenarios" (demand profiles) and solving a *deterministic* optimization problem over those scenarios instead. We refer to the deterministic scenario-based equivalent of the two-stage stochastic inventory management problem described above as "STOC". The STOC formulation is routinely used to solve a variety of stochastic optimization problems. For the distributions of instances we consider in this paper, there is no general (heuristic) recipe for finding feasible solutions. This makes for an interesting learning problem: can our approach discover an effective heuristic through exposure to a set of training instances?

The following mathematical formulation represents STOC:

$$\begin{aligned} \underset{x,y}{\text{maximize}} \quad & \sum_{j=1}^{p} p_j x_j + \sum_{i=1}^{k} \sum_{j=1}^{p} r_l^i y_j^i \\ \text{subject to} \quad & A^i x + D^i y^i \leq c^i,\ i = 1, \ldots, k, \\ & x \in \{0,1\}^p, \\ & y^i \in \{0,1\}^p,\ i = 1, \ldots, k. \end{aligned} \tag{5.5}$$

In this formulation, there are k scenarios, x represents the p first-stage decision variables, whereas y^i are the p second-stage decision variables corresponding to the i-th scenario. Notice that the first-stage variables appear in *all* constraints (5.5) with coefficient matrices A^i, whereas the second-stage variables have non-zero coefficients D^i only in scenario i's constraints. STOC instances have $p(k+1)$ variables and pk constraints.

As in [37], we generate STOC instances by setting values for k and p first. The entries in matrices A^i, D^i and objective function coefficients p and r are drawn uniformly at random from $\{-10, -9, \ldots, 9, 10\}$. The right-hand side values c^i are set such that they guarantee the existence of a feasible 0-1 solution to the constraints. Table 5.1provides additional details on the STOC instances we generated.

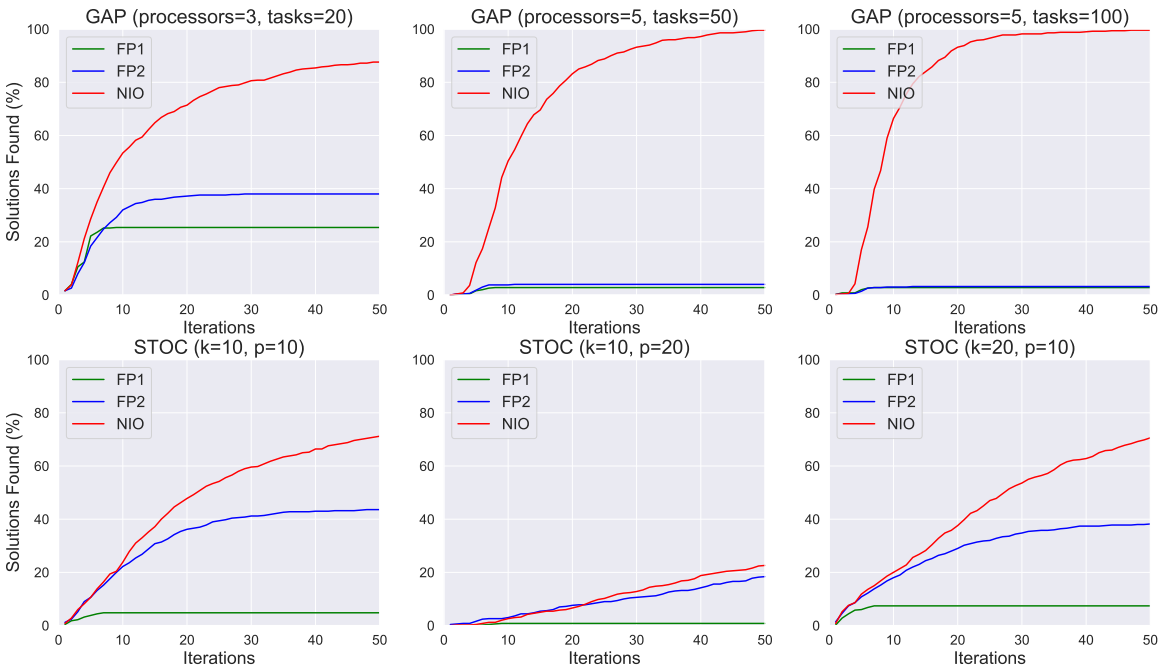


Figure 5.1: Percentage of test instances with feasible solution found (y-axis) as a function of the number of iterations (x-axis).

5.5 Experiments

In this section, we present our experimental setup and results. The training of our NIO models is performed on one of four NVIDIA GTX 1080 GPUs hosted on a server with 128GB of memory and 8 dual-core Intel Core i7-7820X CPUs. Our method is implemented within the PyTorch deep learning framework. For all algorithms we studied, either ECOS or Gurobi were used to solve the linear programming projection on a CPU; Gurobi was only evoked when ECOS struggled with numerical issues. All generated instances were solved to global optimality with Gurobi 8.0.1. We will make our data and code publicly available.

5.5.1 Baselines

We compare our approach to two variants of the “Feasibility Pump". The first variant, FP1, is identical to the original algorithm proposed in Figure 1 of [43]. Starting with the fractional solution to the LP relaxation of the problem, FP1 rounds the solution to the nearest integer point then projects it back into the feasible region, obtaining a new (fractional) solution.

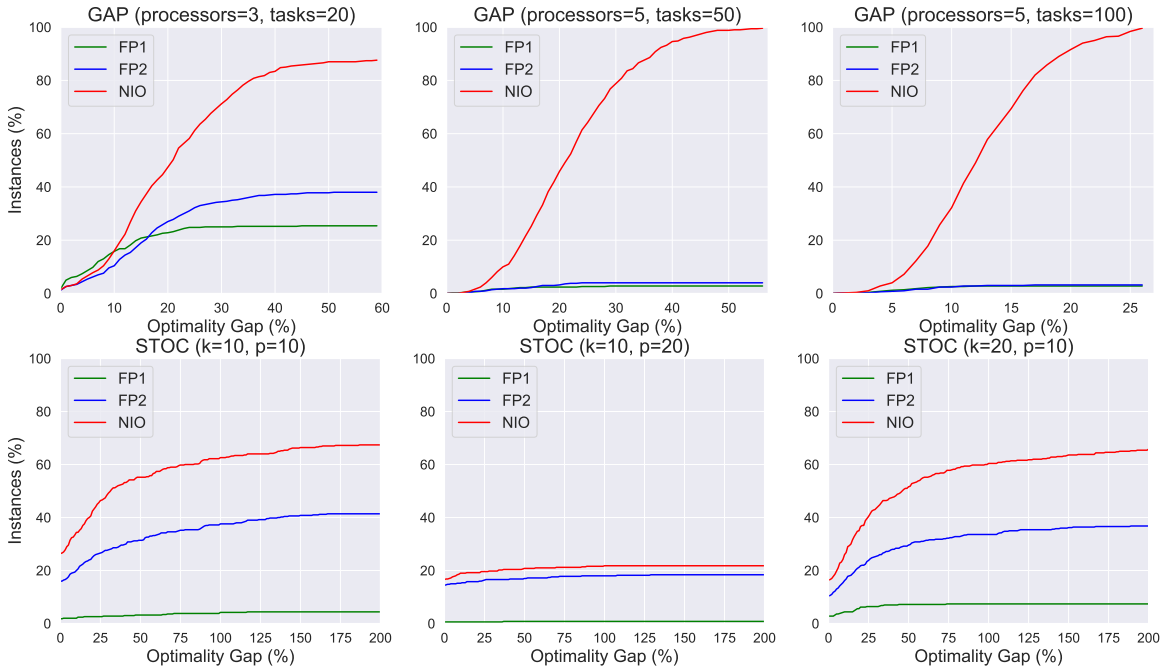


Figure 5.2: The percentage of test instances (y-axis) with optimality gap at most $x\%$ (x-axis).

Whenever the projected point becomes integer, a 0-1 feasible solution to the problem has been found, and the algorithm terminates. If no integer feasible solutions is found in a given maximum round-project iterations, the algorithm fails. Crucially, the projection step consists in solving the following LP:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_{j \in \mathcal{B}: \tilde{x}_j = 0} x_j + \sum_{j \in \mathcal{B}: \tilde{x}_j = 1} (1 - x_j) \\ \text{subject to} \quad & Ax \leq b, \; x \in [0,1]^n. \end{aligned} \tag{5.6}$$

Notice that the objective coefficients are either -1 or 1. Whenever *cycling* occurs, a standard perturbation step described in [43] is used to overcome any stalling.

The second variant of the Feasibility Pump, FP2, proceeds in the same way as FP1, but assigns different objective function coefficients in the projection LP. More specifically,

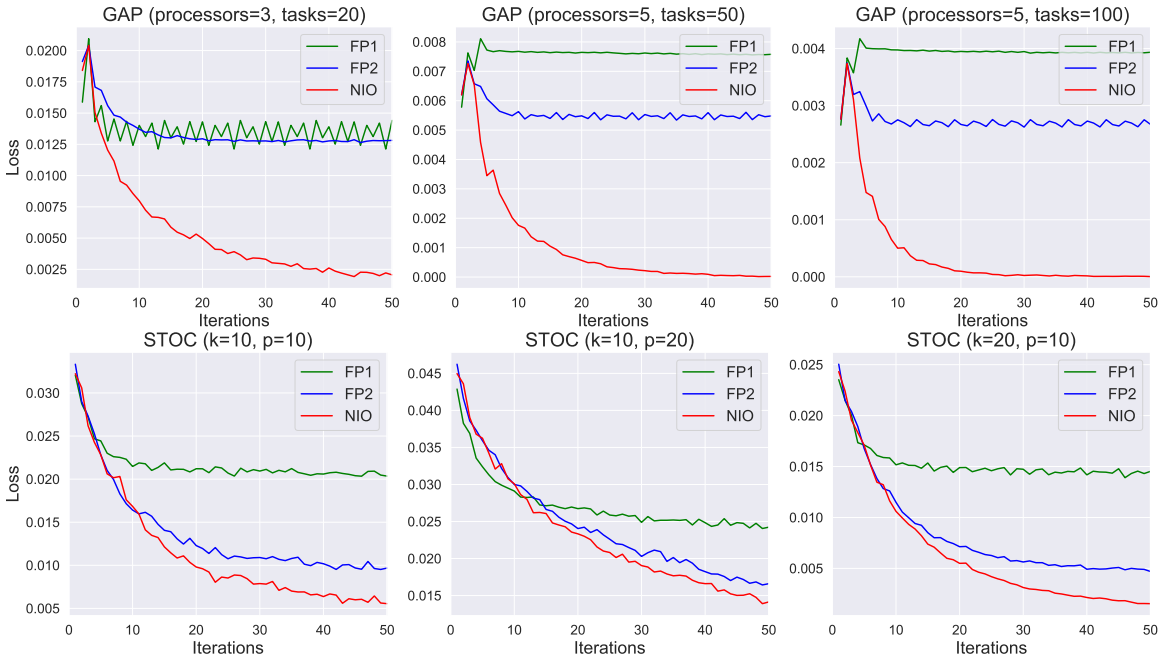


Figure 5.3: The value of the binary cross-entropy loss (y-axis) as a function of the number of iterations (x-axis).

letting $\gamma > 1$, the projection is the following:

$$\begin{aligned} \underset{x}{\text{minimize}} \quad & \sum_{j \in \mathcal{B}: \bar{x}_j = 0} x_j (1 + \gamma \mathbb{I}(\bar{x}_j > 0)) \; + \\ & \sum_{j \in \mathcal{B}: \bar{x}_j = 1} (1 + x_j (-1 - \gamma \mathbb{I}(\bar{x}_j < 1))) \\ \text{subject to} \quad & Ax \leq b, \; x \in [0,1]^n. \end{aligned} \tag{5.7}$$

In words, the projection in FP2 assigns larger coefficients (in magnitude) to variables that are *not integral* in the previous iteration's projected solution, $\bar{x}$. In our experiments, we set $\gamma = 9$: fractional variables then have a coefficient of ± 10 rather than ± 1. We have devised this new projection as a stronger variant of FP1.

5.5.2 Training

We train our NIO models for 50 epochs on 500 training instances processed in mini-batches of size 50 using the Adam optimization algorithm. Each run of NIO consists of 50 iterations at most, terminating as soon as a feasible solution is found. We use "Truncated

Backpropagation Through Time" (TBPTT), unrolling NIO for 5 iterations, a standard practice when training recurrent models [14]. A held-out validation set of 50 instances is used to evaluate the model after each training epoch. For GAP, the training and validation datasets consist of 500 and 50 instances, respectively, each with (3 processors, 20 tasks) and coefficients generated as described in Section 5.4. For STOC, the similarly-sized training and validation datasets have $k = 10$ scenarios and $p = 10$; see Section 5.4 for more details. Adam's learning rate is decayed by a factor of 2 if there is no improvement in the training loss for 5 epochs. We terminate the training early if there is no improvement in the percentage of solutions found for validation instances for 20 consecutive epochs. Table 5.2 lists the hyperparameters considered, the best model's configuration and its validation performance. We found that most hyperparameter configurations resulted in validation performance that is competitive with the best configuration's.

Table 5.2: Hyperparameters of our model and training. Columns "GAP" and "STOC" indicate the best hyperparameter configuration w.r.t. the percentage of instances of the validation set for which a solution was found; the latter value is shown in the bottom row of the table.

Hyperparameter	**Values**	**GAP**	**STOC**
Loss Function	Binary Cross-Entropy (BCE), Product Loss	BCE	BCE
Initial Learning Rate	0.1, 0.01	0.01	0.1
GRU Depth	1, 2	1	2
GRU Hidden Layer Size	8, 16, 32, 64, 128	64	32
	Solutions Found (%) on Validation Set	94	84

5.5.3 Results

For each of GAP and STOC, the corresponding best model described in Table 5.2 is used for testing on unseen instances. All testing sets consist of 500 instances of a given problem; Table 5.1 includes additional information on the number of variables and constraints in instances of each set.

Test Performance

Solutions Found. First, we evaluate NIO's ability to find feasible solutions as compared to FP1 and FP2. In Figure 5.1, we vary the number of iterations of each algorithm along the x-axis and measure the percentage of test instances for which the algorithm has found a feasible solution (and thus terminated) by that iteration.

The leftmost column of Figure 5.1 shows these results for GAP/STOC test sets from the *same distribution as the training sets*, namely GAP (processors = 3, items = 20) and STOC (k = 10, p = 10). NIO manages to find feasible solutions to the majority of test instances in 10 and 20 iterations for GAP and STOC, respectively. In comparison, both FP1 and FP2 find solutions for at most 45% of instances, even when run for the *full 50 iterations*.

Solution Quality. Second, we assess the quality of the solutions found by the algorithms. The standard quality metric is the percentage *optimality gap*, defined as:

$$\text{gap}\,(z^{OPT}, z) = 100 \cdot \frac{|z - z^{OPT}|}{|z^{OPT}| + e^{-10}},$$

with z^{OPT} the optimal value for a given problem instance and z the value of a given solution; a gap of zero implies an optimal solution. For our test instances, the optimal value is computed with the exact solver Gurobi.

We note that none of the three algorithms make use of the objective function coefficients beyond the initial fractional point. Existing tweaks to the projection objective function [3] or constraints [43] can be straightforwardly applied to both NIO and FP1/FP2 to explicitly improve the quality of the solutions.

In Figure 5.2, a point (x, y) is interpreted as follows: the algorithm has found a feasible solution with optimality gap no larger than $x\%$ for $y\%$ of the test instances; an optimal algorithm would appear in the top-left corner of the plot, finding an optimal solution for all instances. For GAP (processors = 3, items = 20) (top row, first column in Figure 5.2), NIO finds good solutions more frequently than FP1 and FP2: NIO's solutions have optimality gaps of at most 20% for around 50% of instances. As for STOC, NIO's solutions have optimality gaps of at most 20% for around 40% of the instances. In comparison, not only

do FP1 and FP2 find solutions for fewer instances than NIO (Figure 5.1), but also fewer high-quality solutions: for example, for STOC (k=10, p=10) (bottom row, first column in Figure 5.2), FP2 finds optimal solutions (0% optimality gap) for less than 20% of test instances , whereas NIOfinds optimal solutions for more than 25% of the test instances.

Consistency of the Loss Function. Does the Binary Cross-Entropy (BCE) loss function used in training NIO models correlate well with our true objective, namely the number of instances with solutions found? In Figure 5.3, we explore this question by plotting the loss function as a function of the number of iterations. While the number of instances with solutions found is monotonically increasing (see Figure 5.1, the loss may not be. However, Figure 5.3 shows an overall decreasing loss that is consistent with the observations in Figure 5.1. This finding brings some assurance to using a surrogate, differentiable loss such as BCE as an alternative to the discrete, non-differentiable count of instances with feasible solutions found.

Generalization to Larger Problems

We demonstrate the ability of our method to generate NIO models that *generalize to larger size instances* than what the model was trained on. This is important as training time depends on the speed with which we solve repeatedly the LP relaxation of the train instances. Hence, being able to train on smaller (faster to solve) instances and then use the model for larger problems can be very advantageous. The leftmost columns of Figure 5.1 and Figure 5.2 compare FP1/FP2 performance to the performance of NIO on test instances that are from the same distribution as the one on which the NIO model was trained (namely GAP (processors = 3, items = 20) and STOC (k=10, p=10)). The middle and rightmost column of these two figures show the performance of these NIO models on instances of the same problem but of larger size.

We observe that, for GAP (top rows of both figures), the NIO model trained on small instances performs really well on larger instances (in fact better than on the original smaller size), while larger instances significantly degrade the FP1/FP2 performance. For STOC,

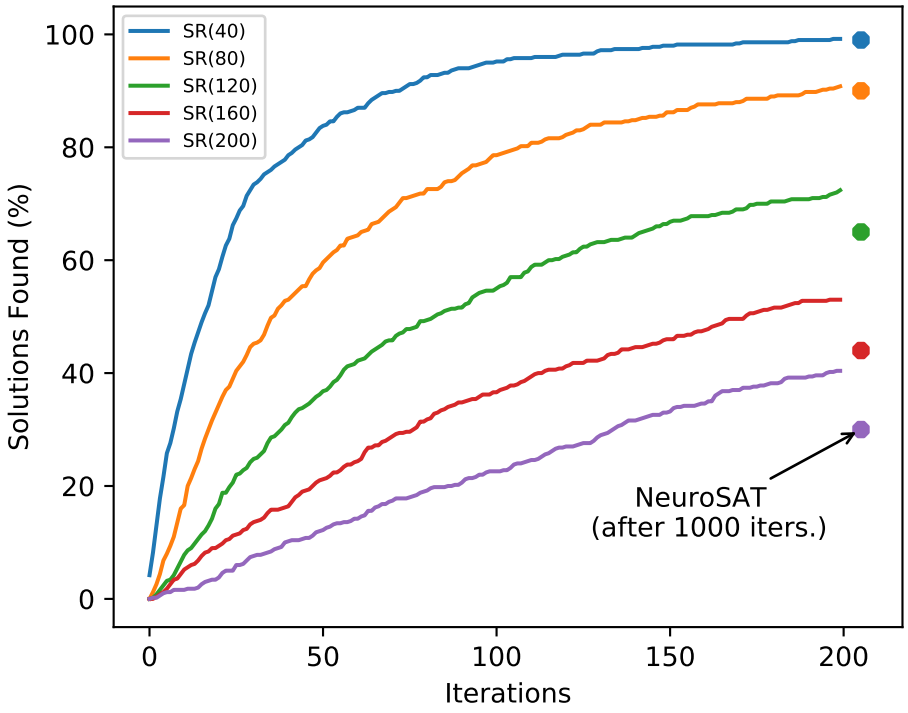


Figure 5.4: Performance of NIO trained over GAP instances and evaluated over SAT instances. The line plot demonstrates the percentage of solutions found over number of instances, where each line belongs to different level of complexity of problem instances. The dots provide a comparative measure of NeuroSAT's performance on solving these instances after 1000 iterations (compared to our 200 iterations.)

when increasing the number of scenarios k (bottom row, leftmost column), the NIO model generalizes well keeping similar performance as on the original distribution of smaller instances. For STOC, when increasing the number of first-stage decisions p (bottom row, middle column), the NIO model does not generalize as well. And although these larger instances become harder to solve also for FP1/FP2, for this setting FP2 almost matches NIO in performance, suggesting that re-training NIO on instances of STOC with the larger p might be warranted.

Generalization Across Problems

Finally, we demonstrate the effectiveness of our method to *generalize across problems* never encountered before in the learning process. Specifically, we generate 500 SAT problem instances of varying complexities, using the code provided by the authors of NeuroSAT [116],

a graph neural network based method tailored for solving SAT instances. This allows us to perform a fair comparison with NeuroSAT model as generated problem instances belong to the same distribution as used in their test. We evaluate the performance of NIO for solving these SAT instances. However, a key consideration in this evaluation is that we never train our model on any SAT instances, instead use our best learned model over the GAP instances described in Section 5.1.

Figure 5.4 provides the performance comparison with original NeuroSAT model in solving these instances (based on the values reported in Figure 5 of [116]) and we highlight three superior qualities of our framework:

1. NIO consistently and significantly outperforms NeuroSAT in terms of finding correct solutions for the test problem instances demonstrating its ability to generalize across problems.

2. It is important to note that NIO only trains over hundreds of GAP instances while NeuroSAT requires training over millions of SAT instances to achieve state-of-the-art performance. This demonstrates the ability of NIO to retain better sample efficiency and generalize to new problems without the need of compromising training efficiency.

3. In Figure 5.4, the circles measure the percentage of instances solved by NueroSat after 1000 iterations. It is remarkable to observe that NIO consistently performs better while running for an order of magnitude of less number of iterations, thereby showcasing its ability to adapt well to new problems.

5.6 Conclusion

We introduced NIO, a neural integer optimization framework that learns heuristics for discrete optimization problems subject to generic constraints. We build an elegant deep recurrent architecture to facilitate this task and make a compelling case of drawing inspiration from techniques in deep learning and machine learning to build state of the art solvers

for discrete optimization problems. Concretely, we propose the idea of incorporating a projection operator into recurrent neural network to build the capacity to generate solutions to discrete optimization problems under complex and generic constraints regime. Our framework is able to train in a simple end-to-end manner unlike existing approaches that either depend on strong supervision or employ complex reinforcement learning techniques limited to specific problem settings. We apply our framework to learn heuristics for two intricate problems in discrete optimization, namely, Generalized Assignment Problem and Stochastic Programming and demonstrate the ability of NIO to significantly outperform widely adopted variants of feasibility pump methods to solve these problems. NIO has been shown to exhibit highly desirable training sample complexity and it generalizes well to large number of problem instances even after training on a small set.

Our work takes a significant stride in establishing that neural network based approaches to learn heuristics has the capacity to generalize across problem domains and have demonstrated its ability to outperform sophisticated methods tailored for specific problem settings. While surprising, this is an extremely encouraging outcome of our work and we believe that it provides a significant next step in advancing the community's evolving understanding on connecting deep learning with discrete optimization, thereby opening several potential future research directions.

Part III

Discrete Optimization for Machine Learning

CHAPTER 6

COMBINATORIAL ATTACKS ON BINARIZED NEURAL NETWORKS

Binarized Neural Networks (BNNs) have recently attracted significant interest due to their computational efficiency. Concurrently, it has been shown that neural networks may be overly sensitive to “attacks" – tiny adversarial changes in the input – which may be detrimental to their use in safety-critical domains. Designing attack algorithms that effectively fool trained models is a key step towards learning robust neural networks. The discrete, non-differentiable nature of BNNs, which distinguishes them from their full-precision counterparts, poses a challenge to gradient-based attacks. In this work, we study the problem of attacking a BNN through the lens of combinatorial and integer optimization. We propose a MIP formulation of the problem. While exact and flexible, the MIP quickly becomes intractable as the network and perturbation space grow. To address this issue, we propose **`IProp`**, a decomposition-based algorithm that solves a sequence of much smaller MIP problems. Experimentally, we evaluate both proposed methods against the standard gradient-based attack (PGD) on MNIST and Fashion-MNIST, and show that **`IProp`** performs favorably compared to PGD, while scaling beyond the limits of the MIP.

6.1 Introduction

The success of neural networks in vision, text and speech tasks has led to their widespread deployment in commercial systems and devices. However, these models can often be fooled by minimal perturbations to their inputs, posing serious security and safety threats [52]. A great deal of current research addresses the “robustification" of neural networks using adversarially generated examples [84, 92], a variant of standard gradient-based training that uses adversarial training examples to defend against possible attacks. Recent work has also formulated the problem of “adversarial learning” as a robust optimization

problem [92, 82, 117], where one seeks the best model parameters with respect to the loss function as measured on the worst-case adversarial perturbation of each point in the training dataset. Attack algorithms may thus be used to augment the training dataset with adversarial examples during training, resulting in more robust models [84]. These new advances further motivate the need to develop effective methods for generating adversarial examples for neural networks.

In this work, we focus on designing effective attacks against *Binarized* Neural Networks (BNNs) [32]. BNNs are neural networks with weights in $\{-1, +1\}$ and the sign function non-linearity, and are especially pertinent in low-power or hardware-constrained settings, where they have the potential to be used at an unprecedented scale if deployed to smartphones and other edge devices. This makes attacking, and consequently robustifying BNNs, a task of major importance. However, the discrete, non-differentiable structure of a BNN renders less effective the typical attack algorithms that rely on gradient information. As strong attacks are crucial to effective adversarial training, we are motivated to address this problem in the hope of generating better attacks.

The goal of adversarial attacks is to modify an input slightly, so that the neural network predicts a different class than what it would have predicted for the original input. More formally, the task of generating an optimal adversarial example is the following:

Given:

– A (clean) data point $x \in \mathbb{R}^n$;

– A trained BNN model with parameters w, that outputs a value $f_c(x; w)$ for a class $c \in \mathcal{C}$;

– `prediction`, the class predicted for data point x, $\arg\max_{c \in \mathcal{C}} f_c(x; w)$;

– `target`, the class we would like to predict for a slightly perturbed version of x;

– ϵ, the maximum amount of perturbation allowed in any of the n dimensions of the input x.

Find:

A point $x' \in \mathbb{R}^n$, such that $\|x - x'\|_\infty \leq \epsilon$ and the following objective function is maximized:

$$\boldsymbol{f}_{\texttt{target}}(\boldsymbol{x'}; \boldsymbol{w}) - \boldsymbol{f}_{\texttt{prediction}}(\boldsymbol{x'}; \boldsymbol{w}).$$

This objective function guides *targeted* attacks [84], and is commonly used in the adversarial learning literature. If an adversary wants to fool a trained model into predicting that an input belongs to a given class, they will simply set the value of `target` accordingly to that given class. We note that our formulation and algorithm also work for *untargeted* attacks via a simple modification of the objective function.

Towards designing *optimal* attacks against BNNs, we propose to model the task of generating an adversarial perturbation as a Mixed Integer Linear Program (MIP). Integer programming is a flexible, powerful tool for modeling optimization problems, and state-of-the-art MIP solvers have achieved excellent results in recent years due to algorithmic and hardware improvements [7]. Using a MIP model is conceptually and practically useful for numerous reasons. First, the MIP is a natural model of the BNN: given that a BNN uses the sign function as activation, the function the network represents is *piecewise constant*, and thus directly representable using linear inequalities and binary variables. Second, the flexibility of MIP allows for various constraints on the type of attacks (e.g. locality as in an early version of [122]), as well as various or even multiple objectives (e.g. minimizing perturbation while maximizing misclassification). Third, globally optimal perturbations can be computed using a MIP solver on small networks, allowing for a precise evaluation of existing attack heuristics in terms of the quality of the perturbations they produce.

The generality and optimality provided by MIP solvers does, however, come at a computational cost. While we were able to solve the MIP to optimality for small networks and perturbation budgets, the solver did not scale much beyond that. Nevertheless, experimental results on small networks revealed a gap between the performance of the gradient-based attack and the best achievable. This finding, coupled with the non-differentiable nature of

the BNN, suggests an alternative: a combinatorial algorithm that is: (a) more scalable than a MIP solve, and (b) more suitable for a non-differentiable objective function.

To this end, we propose **IProp** (Integer Propagation), an attack algorithm that exploits the discrete structure of a BNN, as does the MIP, but is substantially more efficient. **IProp** tunes the perturbation vector by iterations of "target propagation": starting at a desirable activation vector in the last hidden layer D (i.e. a target), **IProp** searches for an activation vector in layer $(D-1)$ that can induce the target in layer D. The process is iterated until the input layer is reached, where a similar problem is solved in continuous perturbation space in order to achieve the first hidden layer's target. Central to our approach is the use of MIP formulations to perform layer-to-layer target propagation. **IProp** is fundamentally novel in two ways:

– To our knowledge, **IProp** is the first target propagation algorithm used in adversarial machine learning, in contrast to the typical use cases of training or credit assignment in neural networks [85, 20];

– The use of exact integer optimization methods within target propagation is also a first, and a promising direction suggested recently in [47].

We evaluate the MIP model, **IProp** and the Projected Gradient Descent method (with restarts) (PGD) [92] – a representative gradient-based attack – on BNN models pre-trained on the MNIST [86] and Fashion-MNIST [126] datasets. We show that **IProp** compares favorably against PGD on a range of networks and across a set of evaluation metrics, especially with small perturbation budgets. As such, we believe that our work is a testament to the promise of integer optimization methods in adversarial learning and discrete neural networks.

This paper is organized as follows: we describe related work in Section 6.2, the MIP formulation in Section 6.3, the heuristic **IProp** in Section 6.4 and experimental results in Section 6.5. We conclude with a discussion on possible avenues for future work in

Section 6.6.

6.2 Related Work

Neural networks with the threshold (sign) activation function date back to early work on the Perceptron. However, the work of [32] revived the interest in Binarized Neural Networks as a computationally cheap alternative to full-precision neural networks. This resurgence is due to an effective training algorithm for BNNs. Since then, BNNs have been used in computer vision [109] and high-performance neural networks [124, 11], among other domains. Notably, BNNs are amenable to extremely fast (embedded) hardware implementations (e.g. as in [95]), which may not be possible even for small full-precision networks.

Adversarial attacks against modern neural networks were first investigated in [26, 121]. Since then, the area of "adversarial machine learning" has developed considerably. In [121], a L-BFGS method is used to find a perturbation of an input that leads to a misclassification. As an efficient alternative to L-BFGS, the Fast Gradient Sign Method (FGSM) was proposed in [52]: FGSM uses the gradient of the loss function with respect to the input to maximize the loss, a cheap operation thanks to backpropagation. Soon thereafter, Projected Gradient Descent (PGD), an iterative variant of FGSM, was shown to produce much more effective attacks [84, 92]; PGD with random restarts is the method that we will compare against in this work. Additionally, the Appendix includes a comparison of the proposed method with SPSA [123]. Other attacks have been developed for different constraints on the allowed amount of perturbation (L_0, L_1, L_2 norms, etc.) [30, 105, 98].

Of relevance to our MIP approach are the MIP attacks against rectified linear unit (ReLU) networks of [122] and [44]. In contrast to binarized networks, ReLU networks are differentiable almost everywhere and thus straightforwardly amenable to attacks via PGD. [48] perform an empirical evaluation of existing attack methods against BNNs and find that BNNs are more robust to gradient-based attacks than their full-precision counterparts.

This finding suggests the search for more powerful attacks that exploit the discrete nature of a BNN, a key motivation for our work here. Most recently, [99] studied the problem of *verifying* BNNs with satisfiability (SAT) solvers and MIP. In contrast to our *optimization* problem of maximizing the difference in outputs for a pair of classes, verification is a *satisfiability* problem that asks to prove that a network will not misclassify a given point, i.e. there is no objective function. As such, SAT solvers fare better than MIP solvers in BNN verification. Our **IProp** algorithm is complementary to the exact verification methods of [99], as it can be used to quickly find a counterexample perturbation, if one exists, which would help resolve the verification question negatively.

6.3 Integer Programming Formulation

We briefly introduce our Mixed Integer Linear Programming formulation for the BNN attack problem. As mentioned earlier, the MIP may not be scalable, but it offers insights into designing better algorithms for our problem, as is the case with our **IProp** algorithm. We operate on a trained, fully-connected, feed-forward BNN with weights $w_{l,j',j} \in \{-1, 1\}$ between each neuron j' in the $(l-1)$-st layer and each neuron j in the l-th layer. The BNN performs, at each of its D hidden layers (r neurons per layer), a linear transformation of the input followed by the (element-wise) application of the sign function, where $\texttt{sign}(x)$ is 1 if $x \geq 0$ and -1 otherwise. The output layer consists of a weighted sum of the final hidden layer's activations. In what follows, we use the notation $[D]$ to denote the set of integers from 1 to D, and $[C, D]$ to denote the set of integers from C to D inclusive.

We use the following variables to formulate the BNN attack:

– p_j: the perturbation in feature j, such that the perturbed point is $x + p$; this is a continuous variable, and the only decision variable in our formulation.

– $a_{l,j}$: the pre-activation sum for the j-th neuron in the l-th layer; for the output ($D+1$-st) layer, $a_{D+1,\texttt{target}}$ and $a_{D+1,\texttt{prediction}}$ are equal to the output values $f_{\texttt{target}}(x'; w)$ and

$f_{\texttt{prediction}}(x'; w)$ of the model for the two classes of interest.

– $h_{l,j}$: this is the activation value for the j-th neuron in the l-th layer, i.e. $h_{l,j} = 1$ if $a_{l,j} \geq 0$ and $h_{l,j} = 0$ otherwise. This is the only set of binary variables in our formulation.

In the following MIP formulation, the constraints essentially implement a forward pass in the BNN, from the perturbed input to the output layer. In particular, (6.2) and (6.3) compute the pre-activation sums, (6.4) and (6.5) are big-M constraints that assign the correct activation value h given the pre-activation a, and (6.6) is the perturbation budget constraint. Note that for (6.4) and (6.5), we require the lower and upper bounds $L_{l,j}$ and $U_{l,j}$ on $a_{l,j}$; those bounds are easily calculated given x and ϵ. We implicitly assume that the input is in $[0, 1]^n$, and constrain the perturbed point to be within this range; this is typical for images for example, where pixels in $[0, 255]$ are scaled to $[0, 1]$.

$$\text{max} \quad a_{D+1,\texttt{target}} - a_{D+1,\texttt{prediction}} \tag{6.1}$$

$$\text{subject to} \quad a_{1,j} = \sum_{j'=1}^{n} w_{1,j',j} \cdot (x_{j'} + p_{j'}) \qquad \forall j \in [r] \tag{6.2}$$

$$a_{l,j} = \sum_{j'=1}^{r} w_{l,j',j} \cdot h_{l-1,j'} \qquad \forall l \in [2, D+1], \forall j \in [r] \tag{6.3}$$

$$a_{l,j} \leq U_{l,j} \cdot \frac{(h_{l,j} + 1)}{2} \qquad \forall l \in [D], \forall j \in [r] \tag{6.4}$$

$$a_{l,j} \geq L_{l,j} \cdot \frac{(1 - h_{l,j})}{2} \qquad \forall l \in [D], \forall j \in [r] \tag{6.5}$$

$$p_j \in [-\epsilon, \epsilon] \qquad \forall j \in [n] \tag{6.6}$$

$$h_{l,j} \in \{-1, 1\} \qquad \forall l \in [D], \forall j \in [r] \tag{6.7}$$

$$a_{l,j} \in [L_{l,j}, U_{l,j}] \qquad \forall l \in [D+1], \forall j \in [r] \tag{6.8}$$

In implementing this formulation, we accommodate "batch normalization" [66], which has been shown to be crucial to the effective training of BNNs [32]. We simply use the parameters learned for batch normalization, as well as the mean and variance over the training data, to compute this linear transformation.

6.4 `IProp`: Integer Target Propagation

As we will see in Section 6.5, solving the MIP attack model becomes difficult very quickly. On the other hand, gradient-based attacks such as PGD are efficient (one forward and backward pass per iteration), but not suitable for BNNs: a trained BNN represents a piecewise constant function with an undefined or zero derivative zero at any point in the input space. This same issue arises when training a BNN. There, [32] propose to replace the sign function activation by a differentiable surrogate function g, where $g(x) = x$ if $x \in [-1, 1]$ and $\texttt{sign}(x)$ otherwise. This surrogate function has derivative 1 with respect to x between -1 and 1, and 0 almost everywhere else. As such, during backpropagation, PGD uses the approximate BNN with g as activation, computing its gradient w.r.t. the input vector, and taking an ascent step to maximize the objective (6.1).

However, as we show in Figure 6.1, the gradient used by PGD may not be indicative of the correct ascent direction. Figure 6.1 illustrates the outputs of a BNN (left) and an approximate BNN (right) with 3 hidden layers and 30 neurons per layer, as a single input value is varied in a small range. Clearly, the approximate BNN can behave arbitrarily differently, and gradient information with respect to the input dimension being varied is not very useful for our task.

Motivated by this observation, as well as the limitations of MIP solving, we propose **`IProp`**, a BNN attack algorithm that operates directly on the original BNN, rather than an approximation of it. To gain intuition as to how **`IProp`** works, it is useful to reason about the form of an optimal solution to our problem. In particular, the objective function (6.1) can be expanded as follows:

$$a_{D+1,\texttt{target}} - a_{D+1,\texttt{prediction}} = \sum_{j=1}^{r} (w_{D+1,j,\texttt{target}} - w_{D+1,j,\texttt{prediction}}) \cdot h_{D,j}.$$

Here, the summation is over the r neurons in layer D, and $h_{D,j} \in \{-1, 1\}$ is the activation of neuron j in the last hidden layer D. Clearly, whenever the weights out of a neuron j into the two output neurons of interest are equal, i.e. $w_{D+1,j,\texttt{target}} = w_{D+1,j,\texttt{prediction}}$, the

activation value of that neuron does not contribute to the objective function. Otherwise, if $w_{D+1,j,\texttt{target}} \neq w_{D+1,j,\texttt{prediction}}$, then an *ideal setting* of the activation $h_{D,j}$ would be $+1$ or -1, since this increases the objective function. Applying the same logic to all neurons in hidden layer D, we obtain an *ideal target* activation vector $\overline{T} \in \{-1, 1\}^r$ which maximizes the objective. However, $\overline{T}$ may not be achievable by any perturbation to input x, especially if the perturbation budget ϵ is sufficiently small. As such, **`IProp`** aims at achieving as many of the ideal target activation values as possible, given ϵ.

`IProp` is summarized in pseudocode below. However, we invite the reader to return to the pseudocode following Section 6.4.3, as a lot of the notation is only introduced there.

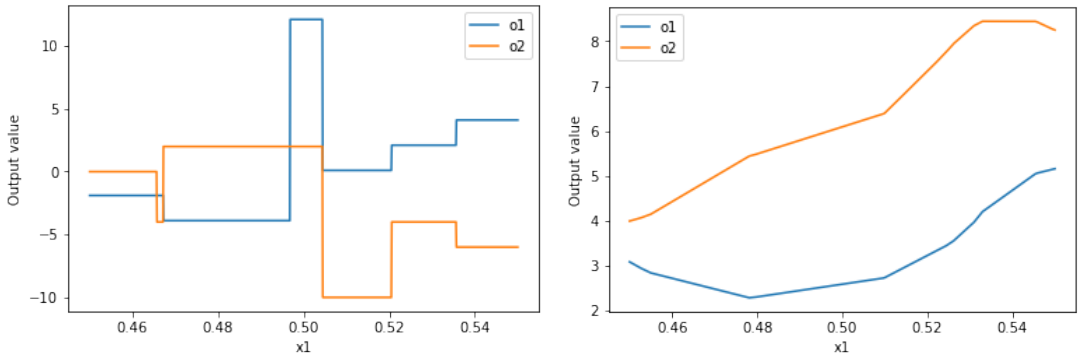


Figure 6.1: Final layer activations for inputs to a small BNN with two output classes (`o1` and `o2`) as a single input dimension (`x1`) is varied. The relative activations of the two classes differ significantly between the true BNN (left) and an approximation of the BNN (right) used to enable gradient computations for PGD.

Algorithm 2 `IProp` (x, ϵ, BNN weight matrices $\{W_l\}_{l=1}^{D}$, `prediction`, `target`, step size S)

Incumbent perturbation: $p^* \leftarrow \mathbf{0}$ (no perturbation)

Compute $\overline{T} \in \{-1, 1\}^r$, the ideal target activation vector in layer D

Run x through BNN; Set h_l^* to resulting activations in layer l for all layers, and $I^* = \{k \in [r] | h_D^*(k) = \overline{T}(k)\}$

$t = 1$

while time limit not reached and not at local optimum **do**

  Sample a set of S neurons $G_D^t \subseteq \{k \in [r] | h_D^*(k) \neq \overline{T}(k)\}$ for layer D

  $T_D^t \coloneqq I^* \cup G_D^t$

  for layer $l = (D-1)$ to 1 **do**

    $T_l^t = \text{argmax}_{h_l \in \{-1,1\}^r} \sum_{j \in T_{l+1}^t} \mathbb{I}\{h_{l+1,j} = T_{l+1}^t(j)\}$ s.t. $h_{l+1} = \text{sign}(W_{l+1} h_l)$

  end for

  $p^t = \text{argmax}_{p \in [-\epsilon, \epsilon]^n} \sum_{j=1}^{r} \mathbb{I}\{h_{1,j} = T_1^t(j)\}$ s.t. $h_1 = \text{sign}(W_1(x+p)), 0 \leq x+p \leq 1$

  if a forward pass with solution $x + p^t$ improves objective (6.1): **then**

    Update incumbent: $p^* \leftarrow p^t$; Update h_l^*, I^*

  end if

  $t = t + 1$

end while

return p^*

6.4.1 Layer-to-Layer Target Satisfaction

Given the ideal target $\overline{T}$, one can ask the following question: how should we set the activation vector T_{D-1}, which consists of the activation values $h_{D-1,j}$ in layer $(D-1)$, such that as much of $\overline{T}$ is achieved after applying the linear transformation and the sign activation? This is a *constraint satisfaction problem* with linear inequalities. More generally, if we would

like a given neuron's activation $h_{l,j}$ to be equal to 1, then the corresponding $a_{l,j}$, defined in (6.3), must be greater than or equal to 0, and vice versa for $h_{l,j}$ to be -1. We cast this binary linear optimization problem as follows:

$$T_l \coloneqq \underset{h_l \in \{-1,1\}^r}{\operatorname{argmax}} \sum_{j=1}^{r} \mathbb{I}\{h_{l+1,j} = T_{l+1}(j)\} \text{ s.t. } h_{l+1} = \text{sign}(W_{l+1} h_l). \tag{6.9}$$

The variables to optimize over in (6.9) are $h_l \in \{-1, 1\}^r$, whereas $T_{l+1} \in \{-1, 1\}^r$ is fixed, as it is provided by the layer $(l + 1)$; we describe this in detail in Section 6.4.2. For instance, when $l = D - 1$ and $T_{l+1} = \overline{T}$, the optimization problem in (6.9) models the satisfaction problem described in the last paragraph.

6.4.2 Target Propagation

Consider solving a sequence of optimization problems based on (6.9), starting with $l = D-1$ and ending with $l = 1$, where each solution T_l to the problem at layer l provides the target for the subsequent problem at layer $(l - 1)$. Then, after obtaining T_1 as a solution to the last optimization problem in the aforementioned sequence, one can search for a perturbation of x that produces T_1, by solving the following mixed binary program:

$$p = \underset{p' \in [-\epsilon, \epsilon]^n}{\operatorname{argmax}} \sum_{j=1}^{r} \mathbb{I}\{h_{1,j} = T_1(j)\} \text{ s.t. } h_1 = \text{sign}(W_1(x + p')), 0 \leq x + p' \leq 1. \tag{6.10}$$

After computing the perturbation p, the point $(x + p)$ is run through the network, and the corresponding objective value (6.1) is computed. The procedure we just described is, at a high-level, a single iteration of our proposed **`IProp`** algorithm. We will describe the full iterative algorithm in Section 6.4.3.

In theory, both optimization problems (6.9) and (6.10) are NP-Hard, by reduction from the MAX-SAT problem, and thus as hard as our MIP problem of Section 6.3. However, in practice, problems (6.9) and (6.10) are much easier to solve than the MIP of Section 6.3, since they are smaller (involving a single hidden layer). We find that for networks with 2-5 hidden layers and 100-500 neurons, these layer-to-layer problems are solved optimally in a few seconds by a MIP solver. It is for this reason that we view **`IProp`** as a *decompo-*

sition algorithm, in that it decomposes the full-network MIP of Section 6.3 into smaller subproblems (6.9) and (6.10).

However, the current description of **IProp** raises two critical questions:

1. When solving problem (6.9) at the last hidden layer, $l = D$, aiming to set $h_{D,j} = T_D(j)$ for *all* neurons may be overly ambitious: if ϵ is very small, then the target propagation is bound to fail when problem (6.10) is solved.

2. In solving the sequence of problems (6.9), a layer l's problem may have multiple optimal solutions that achieve the same number of targets in layer $(l + 1)$. What solutions should we then prefer?

Both of the questions we raised effectively relate to the perturbation budget ϵ: as **IProp** decomposes the attack into layer-to-layer problems (6.9) and (6.10), it is easy to lose track of the global constraint ϵ, which makes many targets T_l impossible to achieve. The solutions that we describe next make **IProp** ϵ-aware, and thus practically effective.

6.4.3 Taking small steps

To address the first question, we take inspiration from gradient optimization methods, which take small steps as determined by a step size (or learning rate), so as to not overshoot good solutions. When solving problem (6.9) at the last hidden layer, we restrict the summation in the objective function to a subset of all neurons; this has the effect of only rewarding target satisfaction up to a limit, so as to not produce overly optimistic solutions that will not withstand the bound ϵ. Specifically, let p^* denote the current incumbent perturbation, initialized to the zero-perturbation vector. Let h_l^* denote the binary activation vector of layer l when the incumbent solution $(x+p^*)$ is run through the BNN. At each iteration t of **IProp**, we solve the sequence of problems (6.9) and then (6.10). To do so, we must specify a set of targets for the first problem (6.9) that is solved at D. This set of targets T_D^t is the union of two sets: the set $I^* = \{k \in [r] | h_D^*(k) = \overline{T}(k)\}$ of already-ideal neurons; and a small

set $G^t \subseteq \{k \in [r] | h_D^*(k) \neq \overline{T}(k)\}$ of neurons who are **not** at their ideal activations under the incumbent. If S denotes the step size, then $|G^t| = S$ for all t. In our implementation, G^t is sampled uniformly and without replacement from all possible S-subsets of non-ideal neurons.

Importantly, after the target T_D^t is specified, target propagation is performed and a potential perturbation p^t is obtained and then run through the BNN. If the objective function (6.1) improves, the incumbent p^* is updated to p^t, and so is the set I^*. In the next iteration, a new target T_D^{t+1} is attempted, and **`IProp`** terminates when it hits a local optimum or runs out of time.

`IProp` is summarized in pseudocode above, with all intermediate optimization problems included, and using common notation.

6.4.4 Maximal Targeting at Minimum Cost

Having presented the full **`IProp`** algorithm, we now address the second question posed at the end of Section 6.4.2: how do we prioritize equally good solutions to problems (6.9)? Intuitively, if two solutions $T_l^{'}$ and $T_l^{''}$ have the same objective value, i.e. satisfy the same number of neurons in layer $(l+1)$, then we would rather use the one which is "closest" to h_l^*, the binary activation vector of layer l under incumbent solution $(x + p^*)$. Such a solution of minimum cost, in the sense of minimum deviation from the forward pass activations of the incumbent, is likely to be easier to achieve when layer $(l-1)$'s problem (6.9) is solved. As a cost metric, we use the L_0 distance between h_l^* and the variables h_l. Note that this cost metric is used as a tie-breaker, and is incorporated into the objective of (6.9) directly with a small multiplier, guaranteeing that the original objective of (6.9) is the first priority. We omit this term from the **`IProp`** pseudocode above for lack of space.

6.5 Experiments

To train the binarized neural networks for which we generate attacks, we use BNN code [1] by [32], and run training experiments on a machine equipped with a GeForce GTX 1080 Ti GPU. We train networks with the following depth x width values: 2x100, 2x200, 2x300, 2x400, 2x500, 3x100, 4x100, 5x100. While these networks are not large by current deep learning standards, they are larger than most networks used in recent papers [44, 99] that leverage integer programming or SAT solving for adversarial attacks or verification. All BNNs are trained to minimize the cross-entropy loss with "batch normalization" [66] for 100 epochs on the full 60,000 MNIST and Fashion-MNIST training images, achieving between 90–95% test accuracy on MNIST, and 80–90% on Fashion-MNIST.

For attack generation, we use the Gurobi Python API to implement and solve our MIP problems, and an implementation of iterated PGD in PyTorch. All methods are run with a time cutoff of 3 minutes on 1,000 test points from the MNIST dataset and 100 test points from the Fashion-MNIST dataset. The MIP problems (6.9), (6.10) solved within **`IProp`** are given a 10 second cutoff. All attacks are run on a cluster of 5 compute nodes, each with 64 cores and 256GB of memory. In the experiments that follow, we specify the class with the second-highest activation (according to the trained model) on the original input as the target class.

6.5.1 Generating Adversarial Examples

Figure 6.2 shows the fraction of MNIST and Fashion-MNIST test points that were flipped by a given attack, for a given network (depth, width) and perturbation budget ϵ; a flip occurs when the objective (6.1) is strictly positive. A higher value is better here. We compare attacks generated using MIP, our method, and PGD on samples from MNIST. For small perturbation budgets ϵ and networks, the MIP approach finds optimal attacks within the time cutoff, but as ϵ and network size grow, solving the MIP becomes increasingly computationally intensive

[1] https://github.com/itayhubara/BinaryNet.pytorch/

and only the best-found solution at timeout is returned. Specifically, for the 2x100 network with $\epsilon = 0.01$, the average runtime of the solver is 27 seconds (all test instances solved to optimality), whereas the same quantity is 777 seconds for the 2x200 network for the same value of ϵ. Similar behavior can be observed as ϵ grows, with most runs timing out at the MIP time limit of 1800 seconds. We believe that this is largely due to the weakness of the linear programming relaxation, as observed by [44], and perhaps the mismatch between the kind of heuristics Gurobi implements versus what would be useful for neural network problems such as ours.

Our method, **IProp** (in red bars), achieves a success rate close to the optimal MIP performance on small networks and ϵ, and scales better than the MIP approach. **IProp** outperforms PGD for nearly all network architectures for the three smaller ϵ values. The better performance of **IProp** compared to PGD is of particular interest for small perturbations, as these are more challenging to detect as attacks. Note that the inputs are in $[0, 1]$, and so $\epsilon = 0.005$ corresponds to a 0.5% change in pixel intensity. For larger values of ϵ, fooling the BNN is relatively easy, as manifested by the high bars. PGD can outperform **IProp** in this easy regime since **IProp** is more computationally expensive. Figure 6.4, shows box plots of the (normalized) objective value (6.1) across the different settings. Consistently with Figure 6.2, **IProp** achieves higher values on average than PGD, indicating that the **IProp** attacks are more effective at modifying the output-layer activations of the networks.

One might wonder about the behavior of the **IProp** and PGD attack methods over time, as PGD is widely regarded as a fast, reasonably-effective attack method. Figure 6.5 shows the relative solution quality over time for each method, averaged over MNIST samples. It is evident that iterated PGD ceases to improve greatly after the first 30 seconds or so. However, more effective attacks are clearly possible, and the **IProp** algorithm constructs progressively stronger attacks that typically surpass the best found PGD attacks after a few more seconds.

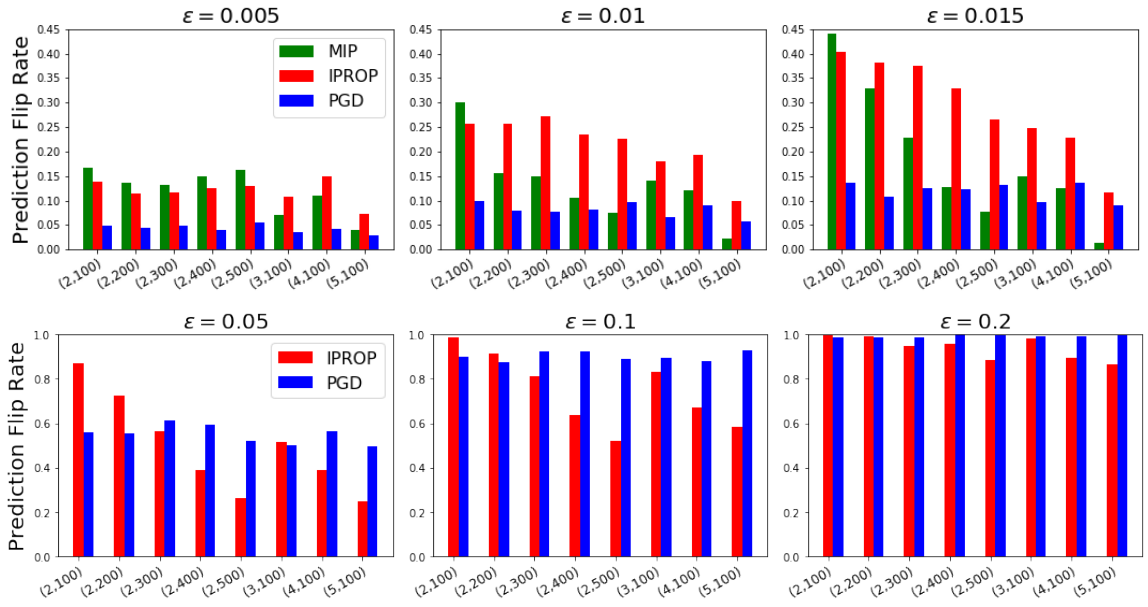


Figure 6.2: Proportion of samples for which the final prediction was flipped to the target class (y-axis) by MIP vs. PGD vs. **IProp** attacks with varying network architectures (x-axis) and varying ϵ (left-right), on the MNIST dataset.

6.5.2 Analysis of **IProp**

Additionally, we investigate the effect of step size S in Line 7 of **IProp** (Figure 6.6). Intuitively, using a small step size S may ensure that the target activations used in each successive iteration are not too difficult to achieve from the current activation in layer D, but this may also lead to multiple iterations and slow improvement over time. Another consideration is that for small perturbation budgets ϵ, large changes in the layer D target activation may propagate back to the first hidden layer, only to fail at the input layer. Meanwhile, wider network architectures may permit the use of larger step sizes. To that end, we devise an *adaptive* step size strategy ("Adaptive", red in all figures): initialized at 5% of the width of the network, the step size S is halved every 5 iterations, if no better incumbent is found. While the hyperparameters of this strategy (initial value, decay rate and number of iterations before decaying) may be optimized, the set of values we used performed reasonably well, as can be seen in Figure 6.6. Indeed, for many of the settings shown, "Adaptive" performs best or close to the best fixed "Constant" step size. Note that previous figures showing **IProp** in red correspond to this very adaptive step size strategy.

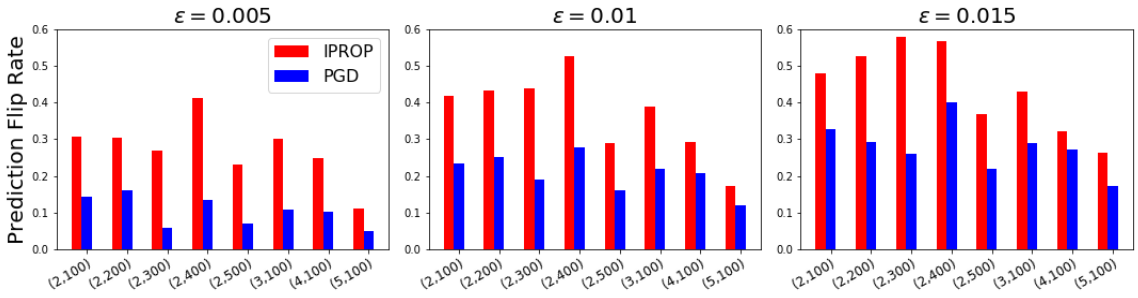


Figure 6.3: Proportion of samples for which the final prediction was flipped to the target class (y-axis) by PGD vs. **IProp** attacks with varying network architectures (x-axis) and varying ϵ (left-right), on the Fashion-MNIST dataset.

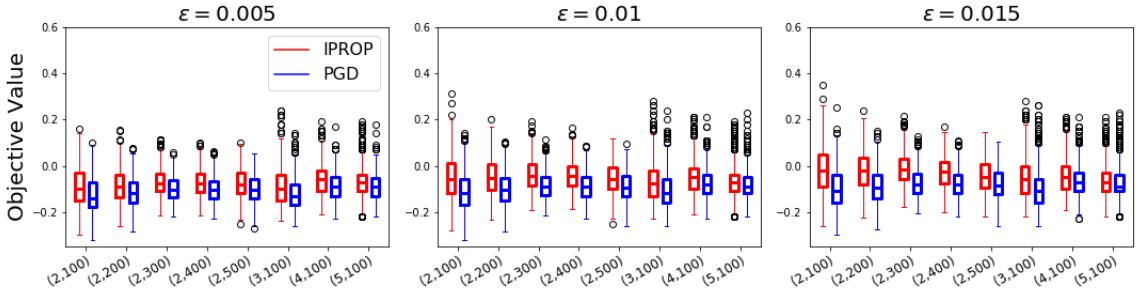


Figure 6.4: Summary statistics for the normalized objective value of attacks obtained by **IProp** versus PGD (y-axis) with varying ϵ in networks with different architectures, on MNIST.

One minor modification that highlights the flexibility of the **IProp** attack method is our ability to warmstart the algorithm with an initial perturbation. For example, we used perturbations obtained by running PGD with a time cutoff of 5 seconds as an alternative to using no perturbation in Line 1 of **IProp**. Figure 6.7 shows that warm starting **IProp** in this manner has the potential to significantly improve the success rate of the resulting attacks, highlighting the value of finding good initial solutions our method, which is essentially a combinatorial local search approach.

6.6 Conclusion & Discussion

We developed combinatorial search methods for generating adversarial examples that fool trained Binarized Neural Networks, based on a Mixed Integer Linear Programming (MIP) model and a target propagation-driven iterative algorithm **IProp**. To our knowledge, this is

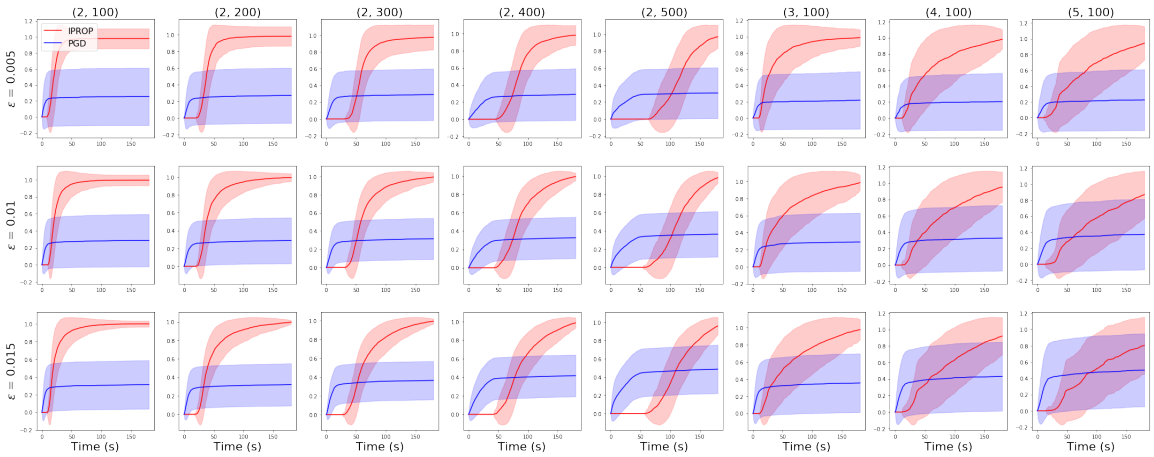


Figure 6.5: Average normalized solution objective value (y-axis) versus runtime (x-axis) for **`IProp`** versus PGD on MNIST samples.

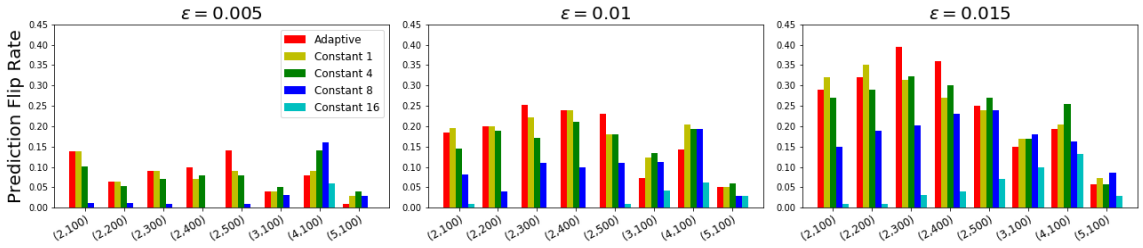


Figure 6.6: Proportion of MNIST samples on which the final prediction was flipped to the target class by **`IProp`** with adaptive or constant step sizes. The adaptive step size performs relatively well across networks of varying size and different values of ϵ.

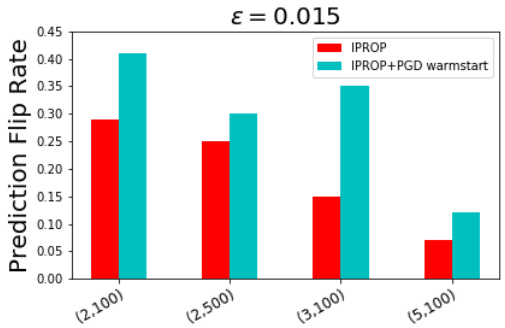


Figure 6.7: Proportion of MNIST samples on which the final prediction was flipped to the target class by **`IProp`** starting with zero perturbation or with an initial perturbation found by running PGD for a short amount of time.

the first such integer optimization-based attack for BNNs, a type of neural networks that is *inherently discrete*. Our MIP model results show that standard (PGD) attack methods often are suboptimal in generating good adversarial examples when the perturbation budget is limited. The ultimate goal is to “attack to protect", i.e. to generate perturbations that can be

used *during adversarial training*, resulting in BNNs that are robust to a class of perturbation. Unfortunately, our MIP model cannot be solved quickly enough to be incorporated into adversarial training. On the other hand, through extensive experiments we have shown that our iterative algorithm **`IProp`** is able to scale-up this solving process while maintaining good performance compared to the PGD attack. With these contributions, we believe we have laid the foundations for improved attacks and potentially robust training of BNNs. This work is a good example of successful cross fertilization of ideas and methods from discrete optimization and machine learning, a growing synergistic area of research, both in terms of using discrete optimization for ML as was done here [47, 24, 25, 13], as well as using ML in discrete optimization tasks [56, 113, 77, 83, 34]. We believe that target propagation ideas such as in **`IProp`** can be potentially extended for the problem of *training* BNNs, a challenging task to this day. The same can be said about hard-threshold networks, as hinted to by [47].

We have identified the following interesting avenues for future work towards attack generation for learning optimally robust BNNs:

– Combinatorial heuristics or approximation algorithms: we believe that the discrete nature of the BNN may allow for the design of principled approximate attacks. Perhaps intelligent search around the LP relaxation solution with forward/backward passes in the network would provide a good heuristic.

– Hybrid methods: fast heuristics such as PGD can be incorporated into the MIP solver as primal heuristics, thus potentially improving the lower bound during search.

– Hyper-parameter optimization: given the complexity of the MIP solver, it is natural to tune its parameters for a set of instances generated from multiple training data points of interest, as is done in [63] with great success. Such a process has the potential to produce parameter settings that result in faster solving times, which may allow for using the MIP solver in the training process.

Part IV

Conclusion

In this dissertation, we studied discrete optimization algorithms through the lens of machine learning. Our primary goal was to improve the performance of discrete optimization algorithms by exploiting the data generated during search and the different instances arising from the same application domain. In the process, we have tackled multiple building blocks of the discrete optimization solving machinery, and developed various machine learning approaches depending on the task and data available. The results of the research contributed towards this goal in the following ways.

First, we showed how exact solvers can benefit from learning at multiple levels. When proving optimality is of primary interest, improving the branching strategy is key. Our key insight here is that the variables of an integer program can be described with a rich set of features, which in turn informs the automated design of novel branching strategies. In an extensive set of experiments, we showed that our supervised ranking approach is capable of learning a branching strategy that outperforms hand-designed, static rules, while being computationally efficient. On the other hand, proving optimality may be of secondary concern as compared to obtaining high-quality feasible solutions early in the search. To alleviate this issue, we propose to learn more intelligent policies for deciding whether a given heuristic should be run at each node of the search tree. Despite the challenging data collection process, where heuristic typically rarely find improved feasible solutions, our simple linear model is capable of better predictions of heuristic success. This translates into more effective branch-and-bound solving: better feasible solutions are found (1) more frequently and (2) earlier in the search, resulting in better pruning by bound and thus faster proofs of optimality. Remarkably, we improve over the solving performance of a state-of-the-art solver, SCIP, whose heuristic selection rules have been fine-tuned by experts over many years. This is true on both a heterogeneous benchmark set of instances as well as a homogeneous family of challenging independent set problems.

Second, we observe that for many real problems, a practitioner may be interested in obtaining high-quality feasible solutions quickly, without necessitating any proofs of

optimality. As such, in a departure from the exact solving setting, we considered the following problem: given a set of problem instances for training, can we learn an effective heuristic that generalizes to unseen instances from the same distribution? For highly structured graph optimization problems, we model construction heuristics as sequential decision policies under the framework of reinforcement learning. This novel formulation lends itself to a natural learning approach, in which an initially random greedy construction policy is tuned via trial-and-error on a set of training instances. However, such an approach is only possible when the action space– here the vertices in a graph –is appropriately represented with predictive features. The key innovation here lies in learning features through a highly effective graph embedding model which can capture complex non-linear interactions between vertices of the graph. The resulting method applies across a wide variety of combinatorial optimization problems on graphs and produces near-optimal greedy heuristics on various synthetic and realistic graph distributions for vertex cover, maximum cut, set cover and the traveling salesman problem. More recently, we identified limitations with the greedy template as it applies to more general discrete problems with complex constraints, such as hard budget constraints (i.e. knapsacks). To address this gap, we proposed the first neural network architecture for learning (non-greedy) heuristics for general integer programs. Our key insight is that a simple “repeated projections" algorithm can be parametrized such that it is possible to tailor it to a given set of training instances. Our neural network architecture combines (1) a recurrent module that can learn useful variable representations over the course of the iterative algorithm with (2) a linear programming module that performs the projection. The training instances are used to tune the parameters of the recurrent module such that it predicts projection directions that result in integer solutions quickly. When applied to generalized assignment and knapsack-like problems, the learned heuristics vastly outperform a widely-used non-learned repeated projection heuristic. In a transfer experiment, we show that a heuristic learned from the assignment problem still performs remarkably well on instances from the satisfiability problem, also outperforming a

recently proposed SAT-specific machine learning approach, while requiring many orders of magnitude less in training instances.

Third, we explored the reverse direction of the fruitful cross-fertilization between machine learning and discrete optimization. We developed combinatorial search methods for generating adversarial examples that fool trained Binarized Neural Networks. To our knowledge, this is the first such integer optimization-based attack for BNNs, a type of neural networks that is inherently discrete. Our MIP model results show that standard gradient-based attack methods often are suboptimal in generating good adversarial examples when the perturbation budget is limited. The ultimate goal is to “attack to protect", i.e. to generate perturbations that can be used during adversarial training, resulting in BNNs that are robust to a class of perturbation. Through extensive experiments, we have shown that our combinatorial algorithm is able to scale-up the attack process while maintaining good performance compared to the gradient attack. With these contributions, we believe we have laid the foundations for improved attacks and potentially robust training of BNNs and other discrete neural network models.

We believe that this dissertation has contributed substantially to the body of knowledge in the area at the intersection of machine learning and discrete optimization. The main driver for this work has been the belief that improved decision-making via computational optimization can bring value to society. The power of appropriately designed machine learning models has been fundamentally transformative in achieving this goal. However, we have taken special care in deeply integrating learning within existing discrete optimization approaches. Such approaches, mostly developed in the Operations Research community, are generally the results of beautiful theoretical and empirical insights, which we have managed to build on and improve over by adopting the perspective of learning-driven algorithm design.

Appendices

APPENDIX A
ADDITIONAL EXPERIMENTS FOR CHAPTER 4

A.1 Set Covering Problem

We also applied our framework to the classical Set Covering Problem (SCP). SCP is interesting because it is not a graph problem, but can be formulated as one. Our framework is capable of addressing such problems seamlessly, as we will show in the coming sections of the appendix which detail the performance of S2V-DQN as compared to other methods.

Set Covering Problem (SCP): Given a bipartite graph G with node set $V \coloneqq \mathcal{U} \cup \mathcal{C}$, find a subset of nodes $S \subseteq \mathcal{C}$ such that every node in $\mathcal{U}$ is covered, i.e. $u \in \mathcal{U} \Leftrightarrow \exists s \in S$ s.t. $(u, s) \in E$, and $|S|$ is minimized. Note that an edge $(u, s), u \in \mathcal{U}, s \in \mathcal{C}$, exists whenever subset s includes element u.

Meta-algorithm: Same as MVC; the termination criterion checks whether all nodes in $\mathcal{U}$ have been covered.

RL formulation: In SCP, the state is a function of the subset of nodes of $\mathcal{C}$ selected so far; an action is to add node of $\mathcal{C}$ to the partial solution; the reward is -1; the termination criterion is met when all nodes of $\mathcal{U}$ are covered; no helper function is needed.

Baselines for SCP: We include *Greedy*, which iteratively selects the node of $\mathcal{C}$ that is not in the current partial solution and that has the most uncovered neighbors in $\mathcal{U}$ [80]. We also used *LP*, another $O(\log |\mathcal{U}|)$-approximation that solves a linear programming relaxation of SCP, and rounds the resulting fractional solution in decreasing order of variable values (SortLP-1 in [108]).

A.2 Experimental Results on Realistic Data

In this section, we show results on realistic nstances for all four problems. In particular, for MVC and SCP, we used the MemeTracker graph to formulate network diffusion optimization problems. For MAXCUT and TSP, we used benchmark instances that arise in physics and transportation, respectively.

A.2.1 Minimum Vertex Cover

As mentioned in the introduction, the MVC problem is related to the efficient spreading of information in networks, where one wants to cover as few nodes as possible such that all nodes have at least one neighbor in the cover. The MemeTracker graph [1] is a network of who-copies-whom, where nodes represent news sites or blogs, and a (directed) edge from u to v means that v frequently copies phrases (or memes) from u. The network is learned from real traces in [51], having 960 nodes and 5000 edges. The dataset also provides the average transmission time $\Delta_{u,v}$ between a pair of nodes, i.e. how much later v copies u's phrases after their publication online, on average. As done in [74], we use these average transmission times to compute a diffusion probability $P(u, v)$ on the edge, such that $P(u, v) = \alpha \cdot \frac{1}{\Delta_{u,v}}$, where α is a parameter of the diffusion model. In both MVC and SCP, we use $\alpha = 0.1$, but results are consistent for other values we have considered. For pairs of nodes that have edges in both directions, i.e. (u, v) and (v, u), we take the average probability to obtain an undirected version of the graph, as MVC is defined for undirected graphs.

Following the widely-adopted Independent Cascade model (see [41] for example), we sample a diffusion cascade from the full graph by independently keeping an edge with probability $P(u, v)$. We then consider the largest connected component in the graph as a single training instance, and train S2V-DQN on a set of such sampled diffusion graphs. The aim is to test the learned model on the (undirected version of the) *full* MemeTracker graph.

Experimentally, an optimal cover has 473 nodes, whereas S2V-DQN finds a cover

[1] http://snap.stanford.edu/netinf/#data

with 474 nodes, only one more than the optimum, at an approximation ratio of 1.002. In comparison, MVCApprox and MVCApprox-Greedy find much larger covers with 666 and 578 nodes, at approximation ratios of 1.408 and 1.222, respectively.

A.2.2 Maximum Cut

A library of Maximum Cut instances is publicly available [2], and includes synthetic and realistic instances that are widely used in the optimization community (see references at library website). We perform experiments on a subset of the instances available, namely ten problems from Ising spin glass models in physics, given that they are realistic and manageable in size (the first 10 instances in Set2 of the library). All ten instances have 125 nodes and 375 edges, with edge weights in $\{-1, 0, 1\}$.

To train our S2V-DQN model, we constructed a training dataset by perturbing the instances, adding random Gaussian noise with mean 0 and standard deviation 0.01 to the edge weights. After training, the learned model is used to construct a cut-set greedily on each of the ten instances, as before.

Table A.1 shows that S2V-DQN finds near-optimal solutions (optimal in 3/10 instances) that are much better than those found by competing methods.

A.2.3 Traveling Salesman Problem

We use the standard TSPLIB library [110] which is publicly available [3]. We target 38 TSPLIB instances with sizes ranging from 51 to 318 cities (or nodes). We do not tackle larger instances as we are limited by the memory of a single graphics card. Nevertheless, most of the instances addressed here are larger than the largest instance used in [19].

We apply S2V-DQN in “Active Search" mode, similarly to [19]: no upfront training phase is required, and the reinforcement learning algorithm 1 is applied on-the-fly on each instance. The best tour encountered over the episodes of the RL algorithm is stored.

[2] http://www.optsicom.es/maxcut/#instances
[3] http://elib.zib.de/pub/mp-testdata/tsp/tsplib/tsp/index.html

Table A.1: MAXCUT results on the ten instances described in A.2.2; values reported are cut weights of the solution returned by each method, where larger values are better (best in bold). Bottom row is the average approximation ratio (lower is better).

Instance	**OPT**	**S2V-DQN**	**MaxcutApprox**	**SDP**
G54100	110	**108**	80	54
G54200	112	**108**	90	58
G54300	106	**104**	86	60
G54400	114	**108**	96	56
G54500	112	**112**	94	56
G54600	110	**110**	88	66
G54700	112	**108**	88	60
G54800	108	**108**	76	54
G54900	110	**108**	88	68
G5410000	112	**108**	80	54
Approx. ratio	1	**1.02**	1.28	1.90

Table A.2 shows the results of our method and six other TSP algorithms. On all but three instances, S2V-DQN finds the best tour among all methods, and is second-best to 2-opt in those three cases. The average approximation ratio of S2V-DQN is also the smallest at 1.05.

A.2.4 Set Covering Problem

The SCP is also related to the diffusion optimization problem on graphs; for instance, the proof of hardness in the classical [73] paper uses SCP for the reduction. As in MVC, we leverage the MemeTracker graph, albeit differently.

We use the same cascade model as in MVC to assign the edge probabilities, and sample graphs from it in the same way. Let $\mathcal{R}^G(u)$ be the set of nodes reachable from u in a sampled graph G. For every node u in G, there are two corresponding nodes in the SCP instance, $u_{\mathcal{C}} \in \mathcal{C}$ and $u_{\mathcal{U}} \in \mathcal{U}$. An edge exists between $u_{\mathcal{C}} \in \mathcal{C}$ and $v_{\mathcal{U}} \in \mathcal{U}$ if and only if $v \in \mathcal{R}^G(u)$. In other words, each node in the sampled graph G has a set consisting of the other nodes that it can reach in G. As such, the SCP reduces to finding the smallest set of nodes whose union can reach all other nodes. We generate training and testing graphs according to this same process, with $\alpha = 0.1$.

Experimentally, we test S2V-DQN and the other baseline algorithms on a set of 1000 test

Table A.2: TSPLIB results: Instances are sorted by increasing size, with the number at the end of an instance's name indicating its size. Values reported are the cost of the tour found by each method (lower is better, best in bold). Bottom row is the average approximation ratio (lower is better).

Instance	OPT	S2V-DQN	2-opt	Cheapest	Christofides	Closest	Nearest	MST
eil51	426	**439**	446	494	527	488	511	614
berlin52	7,542	**7,542**	7,788	9,013	8,822	9,004	8,980	10,402
st70	675	**696**	753	776	836	814	801	858
eil76	538	**564**	591	607	646	615	705	743
pr76	108,159	**108,446**	115,460	125,935	137,258	128,381	153,462	133,471
rat99	1,211	**1,280**	1,390	1,473	1,399	1,465	1,558	1,665
kroA100	21,282	**21,897**	22,876	24,309	26,578	25,787	26,854	30,516
kroB100	22,141	**22,692**	23,496	25,582	25,714	26,875	29,158	28,807
kroC100	20,749	**21,074**	23,445	25,264	24,582	25,640	26,327	27,636
kroD100	21,294	**22,102**	23,967	25,204	27,863	25,213	26,947	28,599
kroE100	22,068	22,913	**22,800**	25,900	27,452	27,313	27,585	30,979
rd100	7,910	**8,159**	8,757	8,980	10,002	9,485	9,938	10,467
eil101	629	**659**	702	693	728	720	817	847
lin105	14,379	**15,023**	15,536	16,930	16,568	18,592	20,356	21,167
pr107	44,303	**45,113**	47,058	52,816	49,192	52,765	48,521	55,956
pr124	59,030	**61,623**	64,765	65,316	64,591	68,178	69,297	82,761
bier127	118,282	**121,576**	128,103	141,354	135,134	145,516	129,333	153,658
ch130	6,110	**6,270**	6,470	7,279	7,367	7,434	7,578	8,280
pr136	96,772	**99,474**	110,531	109,586	116,069	105,778	120,769	142,438
pr144	58,537	**59,436**	60,321	73,032	74,684	73,613	61,652	77,704
ch150	6,528	**6,985**	7,232	7,995	7,641	7,914	8,191	9,203
kroA150	26,524	**27,888**	29,666	29,963	32,631	31,341	33,612	38,763
kroB150	26,130	**27,209**	29,517	31,589	33,260	31,616	32,825	35,289
pr152	73,682	**75,283**	77,206	88,531	82,118	86,915	85,699	90,292
u159	42,080	**45,433**	47,664	49,986	48,908	52,009	53,641	54,399
rat195	2,323	**2,581**	2,605	2,806	2,906	2,935	2,753	3,163
d198	15,780	**16,453**	16,596	17,632	19,002	17,975	18,805	19,339
kroA200	29,368	**30,965**	32,760	35,340	37,487	36,025	35,794	40,234
kroB200	29,437	**31,692**	33,107	35,412	34,490	36,532	36,976	40,615
ts225	126,643	**136,302**	138,101	160,014	145,283	151,887	152,493	188,008
tsp225	3,916	**4,154**	4,278	4,470	4,733	4,780	4,749	5,344
pr226	80,369	**81,873**	89,262	91,023	98,101	100,118	94,389	114,373
gil262	2,378	**2,537**	2,597	2,800	2,963	2,908	3,211	3,336
pr264	49,135	**52,364**	54,547	57,602	55,955	65,819	58,635	66,400
a280	2,579	**2,867**	2,914	3,128	3,125	2,953	3,302	3,492
pr299	48,191	**51,895**	54,914	58,127	58,660	59,740	61,243	65,617
lin318	42,029	45,375	**45,263**	49,440	51,484	52,353	54,019	60,939
linhp318	41,345	45,444	**45,263**	49,440	51,484	52,353	54,019	60,939
Approx. ratio	1	**1.05**	1.09	1.18	1.20	1.21	1.24	1.37

graphs. S2V-DQN achieves an average approximation ratio of 1.001, only slightly behind LP, which achieves 1.0009, and well ahead of Greedy at 1.03.

A.3 Experiment Details

A.3.1 Problem instance generation

Minimum Vertex Cover. For the Minimum Vertex Cover (MVC) problem, we generate random Erdős-Renyi (edge probability 0.15) and Barabasi-Albert (average degree 4) graphs of various sizes, and use the integer programming solver CPLEX 12.6.1 with a time cutoff of 1 hour to compute optimal solutions for the generated instances. When CPLEX fails to find an optimal solution, we report the best one found within the time cutoff as “optimal". All graphs were generated using the NetworkX [4] package in Python. Maximum Cut. For the Maximum Cut (MAXCUT) problem, we use the same graph generation process as in MVC, and augment each edge with a weight drawn uniformly at random from $[0, 1]$. We use a quadratic formulation of MAXCUT with CPLEX 12.6.1. and a time cutoff of 1 hour to compute optimal solutions, and report the best solution found as “optimal". Traveling Salesman Problem.

For the (symmetric) 2-dimensional TSP, we use the instance generator of the 8th DIMACS Implementation Challenge [5] [69] to generate two types of Euclidean instances: “random" instances consist of n points scattered uniformly at random in the $[10^6, 10^6]$ square, while “clustered" instances consist of n points that are clustered into $n/100$ clusters; generator details are described in page 373 of [69].

To compute optimal TSP solutions for both TSP, we use the state-of-the-art solver, Concorde [6] [15], with a time cutoff of 1 hour.

Set Covering Problem. For the SCP, given a number of node n, roughly $0.2n$ nodes are in node-set $\mathcal{C}$, and the rest in node-set $\mathcal{U}$. An edge between nodes in $\mathcal{C}$ and $\mathcal{U}$ exists

[4] https://networkx.github.io/
[5] http://dimacs.rutgers.edu/Challenges/TSP/
[6] http://www.math.uwaterloo.ca/tsp/concorde/

with probability either 0.05 or 0.1, which can be seen as “density" values, and commonly appear for instances used in optimization papers on SCP [17]. We guarantee that each node in $\mathcal{U}$ has at least 2 edges, and each node in $\mathcal{C}$ has at least one edge, a standard measure for SCP instances [17]. We also use CPLEX 12.6.1. with a time cutoff of 1 hour to compute a near-optimal or optimal solution to a SCP instance.

A.3.2 Full results on solution quality

Table A.1 is a complete version of Table 4.2 that appears in the main text.

A.3.3 Full results on generalization

The full generalization results can be found in Table A.3, A.4, A.5, A.6, A.7, A.8 , A.9 and A.10.

Table A.3: S2V-DQN's generalization on MVC problem in ER graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0032	1.0883	1.0941	1.0710	1.0484	1.0365	1.0276	1.0246	1.0111
40-50	\	1.0037	1.0076	1.1013	1.0991	1.0800	1.0651	1.0573	1.0299
50-100	\	\	1.0079	1.0304	1.0570	1.0532	1.0463	1.0427	1.0238
100-200	\	\	\	1.0102	1.0095	1.0136	1.0142	1.0125	1.0103
400-500	\	\	\	\	\	\	1.0021	1.0027	1.0057

Table A.4: S2V-DQN's generalization on MVC problem in BA graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0016	1.0027	1.0039	1.0066	1.0093	1.0106	1.0125	1.0150	1.0491
40-50	\	1.0027	1.0051	1.0092	1.0130	1.0144	1.0161	1.0170	1.0228
50-100	\	\	1.0033	1.0041	1.0045	1.0040	1.0045	1.0048	1.0062
100-200	\	\	\	1.0016	1.0020	1.0019	1.0021	1.0026	1.0060
400-500	\	\	\	\	\	\	1.0025	1.0026	1.0030

A.3.4 Experiment Configuration of S2V-DQN

The node/edge representations and hyperparameters used in our experiments is shown in Table A.11. For our method, we simply tune the hyperparameters on small graphs (i.e., the

Table A.5: S2V-DQN's generalization on MAXCUT problem in ER graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0034	1.0167	1.0407	1.0667	1.1067	1.1489	1.1885	1.2150	1.1488
40-50	\	1.0127	1.0154	1.0089	1.0198	1.0383	1.0388	1.0384	1.0534
50-100	\	\	1.0112	1.0024	1.0109	1.0467	1.0926	1.1426	1.1297
100-200	\	\	\	1.0005	1.0021	1.0211	1.0373	1.0612	1.2021
200-300	\	\	\	\	1.0106	1.0272	1.0487	1.0700	1.1759

Table A.6: S2V-DQN's generalization on MAXCUT problem in BA graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0055	1.0119	1.0176	1.0276	1.0357	1.0386	1.0335	1.0411	1.0331
40-50	\	1.0107	1.0119	1.0139	1.0144	1.0119	1.0039	1.0085	0.9905
50-100	\	\	1.0150	1.0181	1.0202	1.0188	1.0123	1.0177	1.0038
100-200	\	\	\	1.0166	1.0183	1.0166	1.0104	1.0166	1.0156
200-300	\	\	\	\	1.0420	1.0394	1.0290	1.0319	1.0244

Table A.7: S2V-DQN's generalization on TSP in random graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0147	1.0511	1.0702	1.0913	1.1022	1.1102	1.1124	1.1156	1.1212
40-50	\	1.0533	1.0701	1.0890	1.0978	1.1051	1.1583	1.1587	1.1609
50-100	\	\	1.0701	1.0871	1.0983	1.1034	1.1071	1.1101	1.1171
100-200	\	\	\	1.0879	1.0980	1.1024	1.1056	1.1080	1.1142
200-300	\	\	\	\	1.1049	1.1090	1.1084	1.1114	1.1179

Table A.8: S2V-DQN's generalization on TSP in clustered graphs.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0214	1.0591	1.0761	1.0958	1.0938	1.0966	1.1009	1.1012	1.1085
40-50	\	1.0564	1.0740	1.0939	1.0904	1.0951	1.0974	1.1014	1.1091
50-100	\	\	1.0730	1.0895	1.0869	1.0918	1.0944	1.0975	1.1065
100-200	\	\	\	1.1009	1.0979	1.1013	1.1059	1.1048	1.1091
200-300	\	\	\	\	1.1012	1.1049	1.1080	1.1067	1.1112

Table A.9: S2V-DQN's generalization on SCP with edge probability 0.05.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0055	1.0170	1.0436	1.1757	1.3910	1.6255	1.8768	2.1339	3.0574
40-50	\	1.0039	1.0083	1.0241	1.0452	1.0647	1.0792	1.0858	1.0775
50-100	\	\	1.0056	1.0199	1.0382	1.0614	1.0845	1.0821	1.0620
100-200	\	\	\	1.0147	1.0270	1.0417	1.0588	1.0774	1.0509
200-300	\	\	\	\	1.0273	1.0415	1.0828	1.1357	1.2349

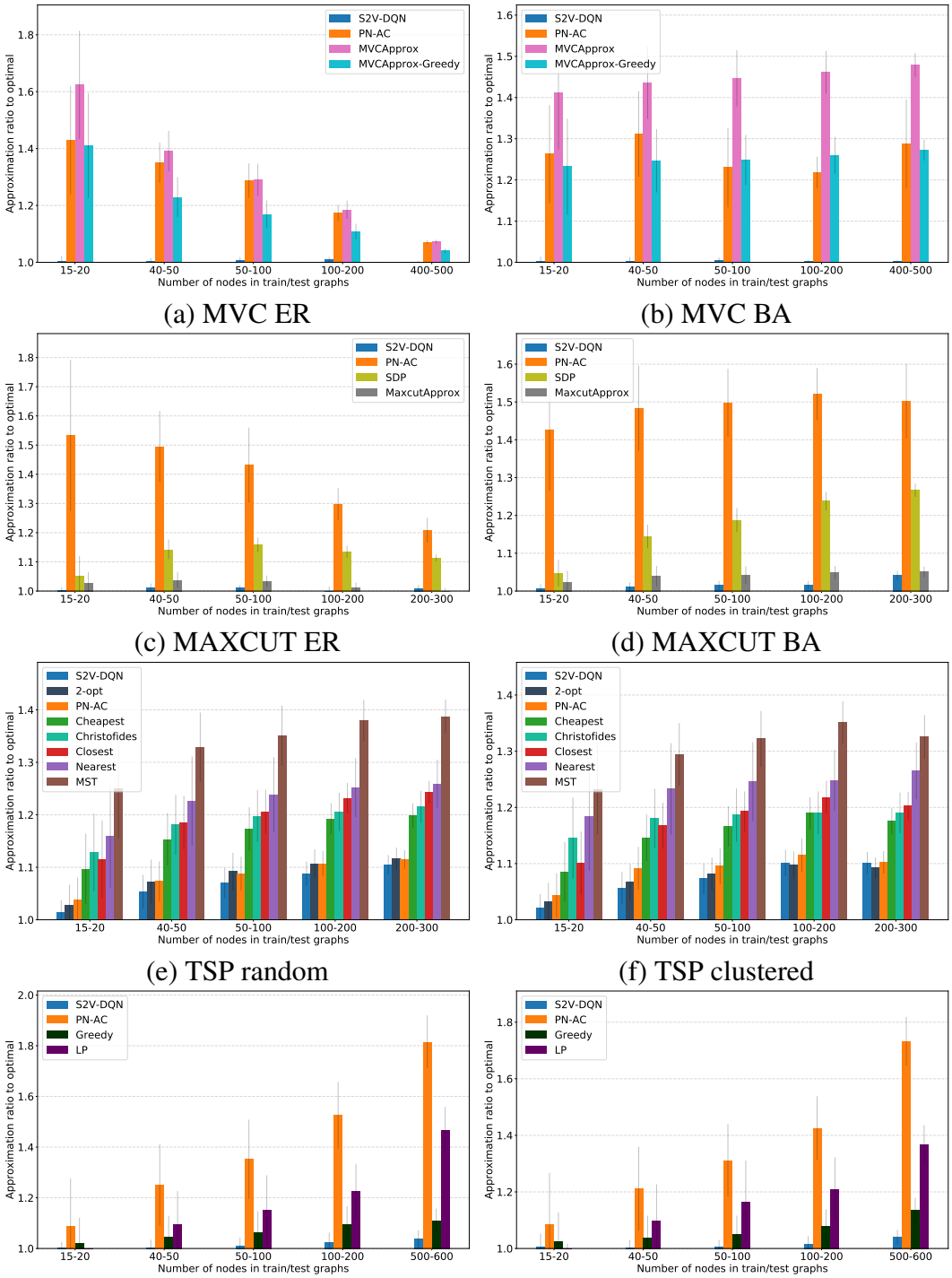


Figure A.1: Approximation ratio on 1000 test graphs. Note that on MVC, our performance is pretty close to optimal. In this figure, training and testing graphs are generated according to the same distribution.

graphs with less than 50 nodes), and fix them for larger graphs.

Table A.10: S2V-DQN's generalization on SCP with edge probability 0.1.

Train \ Test	15-20	40-50	50-100	100-200	200-300	300-400	400-500	500-600	1000-1200
15-20	1.0015	1.0200	1.0369	1.0795	1.1147	1.1290	1.1325	1.1255	1.0805
40-50	\	1.0048	1.0137	1.0453	1.0849	1.1055	1.1052	1.0958	1.0618
50-100	\	\	1.0090	1.0294	1.0771	1.1180	1.1456	1.2161	1.0946
100-200	\	\	\	1.0231	1.0394	1.0564	1.0702	1.0747	2.5055
200-300	\	\	\	\	1.0378	1.0517	1.0592	1.0556	1.3192

Table A.11: S2V-DQN's configuration used in Experiment.

Problem	**Node tag**	**Edge feature**	**Embedding size** p	T	**Batch size**	**n-step**
Minimum Vertex Cover	0/1 tag	N/A	64	5	128	5
Maximum Cut	0/1 tag	edge length; end node tag	64	3	64	1
Traveling Salesman Problem	coordinates; 0/1 tag; start/end node	edge length; end node tag	64	4	64	1
Set Covering Problem	0/1 tag	N/A	64	5	64	2

A.3.5 Stabilizing the training of S2V-DQN

For the learning rate, we use exponential decay after a certain number of steps, where the decay factor is fixed to 0.95. We also anneal the exploration probability ϵ from 1.0 to 0.05 in a linear way.

We also normalize the intermediate reward by the maximum number of nodes. For Q-learning, it is also important to disentangle the actual Q with obsolete $\tilde{Q}$, as mentioned in [97].

Also for TSP with insertion helper function, we find it works better with negative version of designed reward function. This sounds counter intuitive at the beginning. However, since typically the RL agent will bias towards most recent rewards, flipping the sign of reward function suggests a focus over future rewards. This is especially useful with the insertion construction. But it shows that designing a good reward function is still challenging for learning combinatorial algorithm, which we will investigate in our future work.

A.3.6 Convergence of S2V-DQN

In Figure A.2, we plot our algorithm's convergence with respect to the held-out validation performance. We first obtain the convergence curve for each type of problem under every

graph distribution. To visualize the convergence at the same scale, we plot the approximate ratio.

Figure A.2 shows that our algorithm converges nicely on the MVC, MAXCUT and SCP problems. For the MVC, we use the model trained on small graphs to initialize the model for training on larger ones. Since our model also generalizes well to problems with different sizes, the curve looks almost flat. For TSP, where the graph is essentially fully connected, it is harder to learn a good model based on graph structure. Nevertheless, as shown in previous section, the graph embedding can still learn good feature representations with multiple embedding iterations.

A.3.7 Complete time v/s approximation ratio plots

Figure A.3 is a superset of Figure 4.3, including both graph types and three graph size ranges for MVC, MAXCUT and SCP. In this figure A.3, each dot represents a solution found for a single problem instance. For CPLEX, we also record the time and quality of each solution it finds. For example, CPLEX-1st means the first feasible solution found by CPLEX.

A.3.8 Additional analysis of the trade-off between time and approx. ratio

Tables A.12 and A.13 offer another perspective on the trade-off between the running time of a heuristic and the quality of the solution it finds. We ran CPLEX for MVC and MAXCUT for 10 minutes on the 200-300 node graphs, and recorded the time and value of all the solutions found by CPLEX within the limit; results shown next carry over to smaller graphs. Then, for a given method M that terminates in T seconds on a graph G and returns a solution with approximation ratio R, we asked the following 2 questions:

1. If CPLEX is given the same amount of time T for G, how well can CPLEX do?
2. How long does CPLEX need to find a solution of same or better quality than the one the heuristic has found?

For the first question, the column “Approx. Ratio of Best Solution" in Tables A.12 and A.13 shows the following:

– MVC (Table A.12): The larger values for S2V-DQN imply that solutions we find quickly are of higher quality, as compared to the MVCApprox/Greedy baselines.

– MAXCUT (Table A.13): On most of the graphs, CPLEX cannot find any solution at all if given the same time as S2V-DQN or MaxcutApprox. SDP (solved with state-of-the-art CVX solver) is so slow that CPLEX finds solutions that are 10% better than those of SDP if given the same time as SDP (on ER graphs), which confirms that SDP is not time-efficient. One possible interpretation of the poor performance of SDP is that its theoretical guaranteed of 0.87 is *in expectation* over the solutions it can generate, and so the variance in the approximation ratios of these solutions may be very large.

For the second question, the column “Additional Time Needed" in Tables A.12 and A.13 shows the following:

– MVC (Table A.12): The larger values for S2V-DQN imply that solutions we find are harder to improve upon, as compared to the MVCApprox/Greedy baselines.

– MAXCUT (Table A.13): On ER (BA) graphs, CPLEX (10 minute-cutoff) cannot find a solution that is better than those of S2V-DQN or MaxcutApprox on many instances (e.g. the value (59) for S2V-DQN on ER graphs means that on $41 = 100 - 59$ graphs, CPLEX could not find a solution that is as good as S2V-DQN’s). When we consider only those graphs for which CPLEX could find a better solution, S2V-DQN’s solutions take significantly more time for CPLEX to beat, as compared to MaxcutApprox and SDP. The negative values for SDP indicate that CPLEX finds a solution better than SDP’s in a shorter time.

Table A.12: Minimum Vertex Cover (100 graphs with 200-300 nodes): Trade-off between running time and approximation ratio. An “Approx. Ratio of Best Solution" value of 1.x% means that the solution found by CPLEX if given the same time as a certain heuristic (in the corresponding row) is x% worse, on average. “Additional Time Needed" in seconds is the additional amount of time needed by CPLEX to find a solution of value at least as good as the one found by a given heuristic; negative values imply that CPLEX finds such solutions faster than the heuristic does. Larger values are better for both metrics. The values in parantheses are the number of instances (out of 100) for which CPLEX finds some solution in the given time (for “Approx. Ratio of Best Solution"), or finds some solution that is at least as good as the heuristic’s (for “Additional Time Needed").

	Approx. Ratio of Best Solution		Additional Time Needed	
	ER	BA	ER	BA
S2V-DQN	1.09 (100)	1.81 (100)	2.14 (100)	137.42 (100)
MVCApprox-Greedy	1.07 (100)	1.44 (100)	1.92 (100)	0.83 (100)
MVCApprox	1.03 (100)	1.24 (98)	2.49 (100)	0.92 (100)

Table A.13: Maximum Cut (100 graphs with 200-300 nodes): please refer to the caption of Table A.12.

	Approx. Ratio of Best Solution		Additional Time Needed	
	ER	BA	ER	BA
S2V-DQN	N/A (0)	1081.45 (1)	8.99 (59)	402.05 (34)
MaxcutApprox	1.00 (48)	340.11 (3)	-0.23 (50)	218.19 (57)
SDP	0.90 (100)	0.84 (100)	-6.06 (100)	-5.54 (100)

A.3.9 Visualization of solutions

In Figure A.4, A.5 and A.6, we visualize solutions found by our algorithm for MVC, MAXCUT and TSP problems, respectively. For the ease of presentation, we only visualize small-size graphs. For MVC and MAXCUT, the graph is of the ER type and has 18 nodes. For TSP, we show solutions for a “random" instance (18 points) and a “clustered" one (15 points).

For MVC and MAXCUT, we show two step by step examples where S2V-DQN finds the optimal solution. For MVC, it seems we are picking the node which covers the most edges in the current state. However, in a more detailed visualization in Appendix A.3.10, we show that our algorithm learns a smarter greedy or dynamic programming like strategy. While picking the nodes, it also learns how to keep the connectivity of graph by scarifying the intermediate edge coverage a little bit.

In the example of MAXCUT, it is even more interesting to see that the algorithm did not pick the node which gives the largest intermediate reward at the beginning. Also in the intermediate steps, the agent seldom chooses a node which would cancel out the edges that are already in the cut set. This also shows the effectiveness of graph state representation, which provides useful information to support the agent's node selection decisions. For TSP, we visualize an optimal tour and one found by S2V-DQN for two instances. While the tours found by S2V-DQN differ slightly from the optimal solutions visualized, they are of comparable cost and look qualitatively acceptable. The cost of the tours found by S2V-DQN is within 0.07% and 0.5% of optimum, respectively.

A.3.10 Detailed visualization of learned MVC strategy

In Figure A.7, we show a detailed comparison with our learned strategy and two other simple heuristics. We find that the S2V-DQN can learn a much smarter strategy, where the agent is trying to maintain the connectivity of graph during node picking and edge removal.

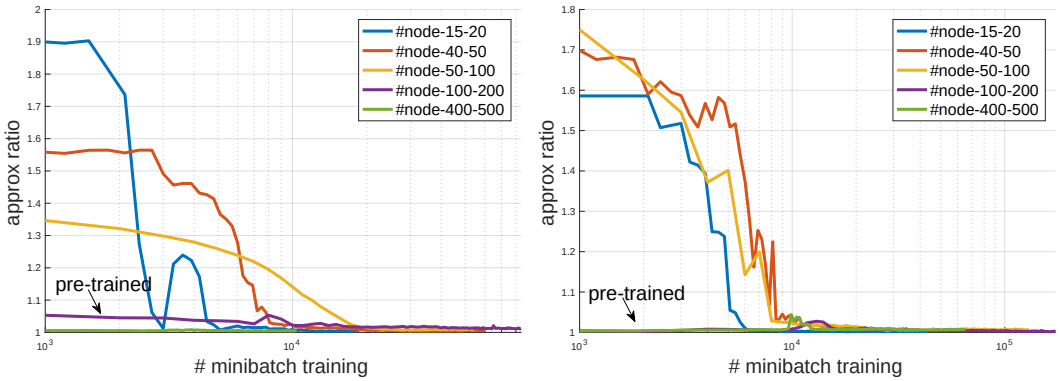
A.3.11 Experiment Configuration of PN-AC

We implemented PN-AC to the best of our capabilities. Note that it is quite possible that there are minor differences between our implementation and [19] that might have resulted in performance not as good as reported in that paper.

For experiments of PN-AC across all tasks, we follow the configurations provided in [19]: *a*) For the input data, we use mini-batches of 128 sequences with 0-paddings to the maximal input length (which is the maximal number of nodes) in the training data. *b*) For node representation, we use coordinates for TSP, so the input dimension is 2. For MVC, MAXCUT and SCP, we represent nodes based on the adjacency matrix of the graph. To get a fixed dimension representation for each node, we use SVD to get a low-rank approximation of the adjacency matrix. We set the rank as 8, so that each node in the input sequence is represented by a 8-dimensional vector. *c*) For the network structure, we use standard single-layer LSTM cells with 128 hidden units for both encoder and decoder parts of the pointer networks. *d*) For the optimization method, we train the PN-AC model with the Adam optimizer [79] and use an initial learning rate of 10^{-3} that decay every 5000 steps by a factor of 0.96. *e*) For the glimpse trick, we exactly use one-time glimpse in our implementation, as described in the original PN-AC paper. f) We initialize all the model parameters uniformly random within $[-0.08, 0.08]$ and clip the $L2$ norm of the gradients to 1.0. g) For the baseline function in the actor-critic algorithm, we tried the critic network in our implementation, but it hurts the performance according to our experiments. So we use the exponential moving average performance of the sampled solution from the pointer network as the baseline.

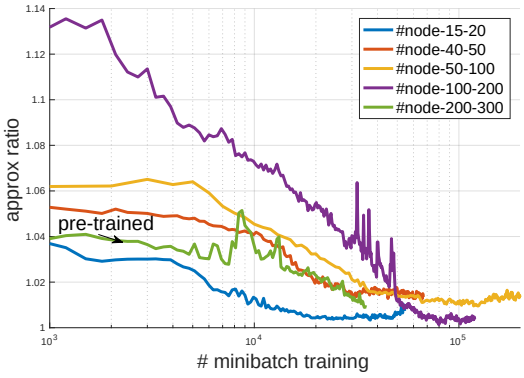
Consistency with the results from [19] Though our TSP experiment setting is not exactly the same as [19], we still include some of the results directly here, for the sake of completeness. We applied the insertion heuristic to PN-AC as well, and all the results reported in our paper are with the insertion heuristic. We compare the approximation ratio reported by [19] verses which reported by our implementation. For TSP20: 1.02 vs 1.03 (reported in our paper); TSP50: 1.05 vs 1.07 (reported in our paper); TSP100: 1.07 vs 1.09

(reported in our paper). Note that we have variable graph size in each setting (where the original PN-AC is only reported on fixed graph size), which makes the task more difficult. Therefore, we think the performance gap here is pretty reasonable.

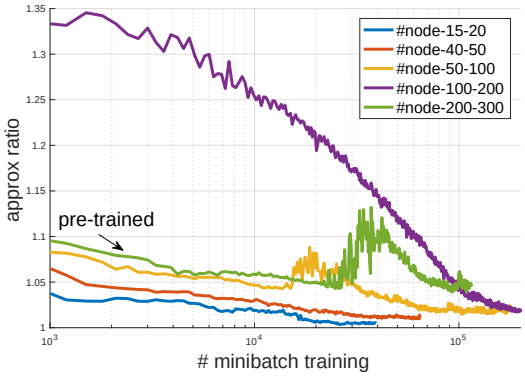


(a) MVC ER

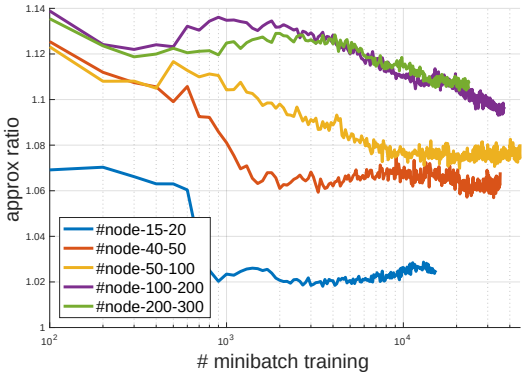
(b) MVC BA



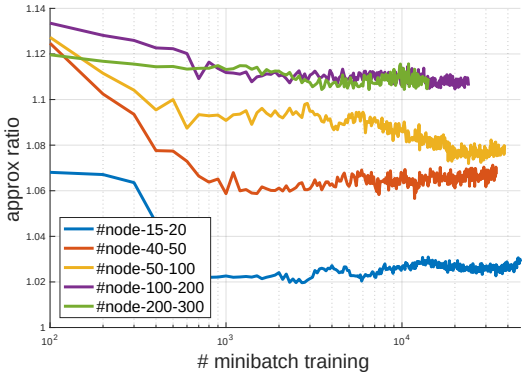
(c) MAXCUT ER



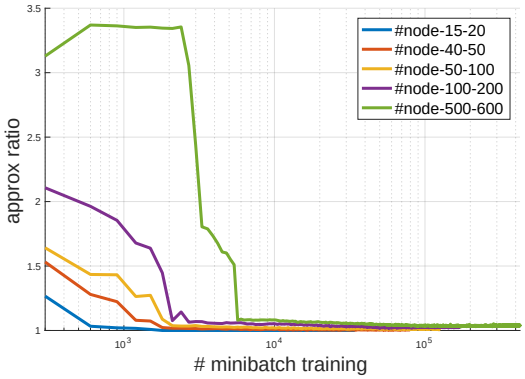
(d) MAXCUT BA



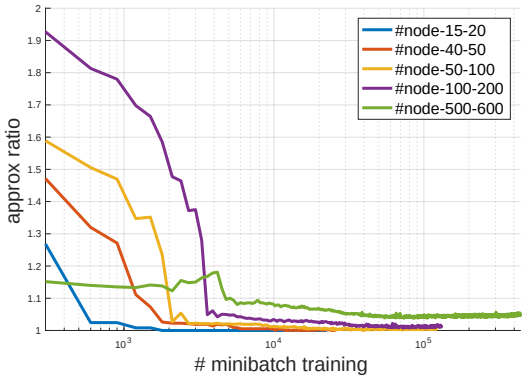
(e) TSP random



(f) TSP clustered

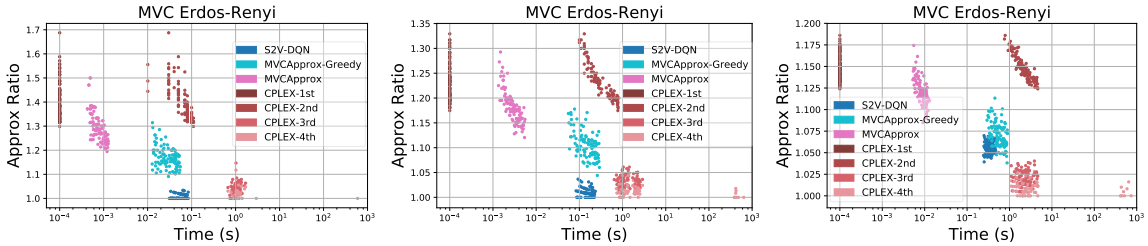


(g) SCP 0.1

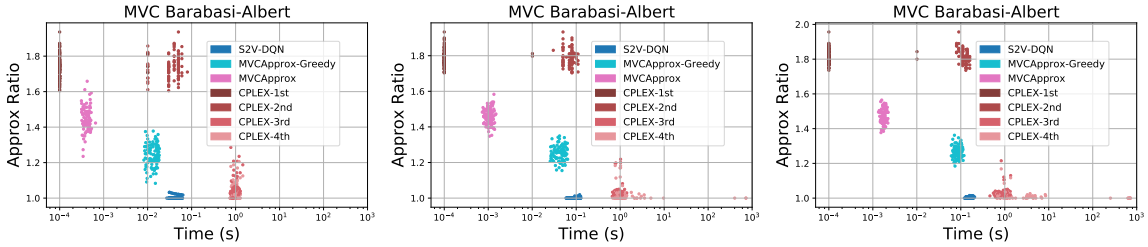


(h) SCP 0.05

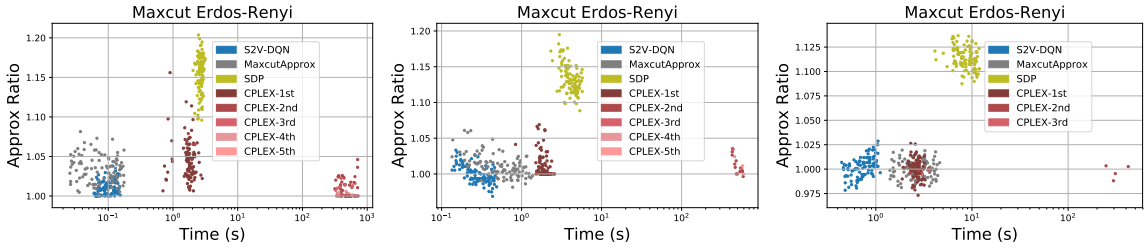
Figure A.2: S2V-DQN convergence measured by the held-out validation performance.



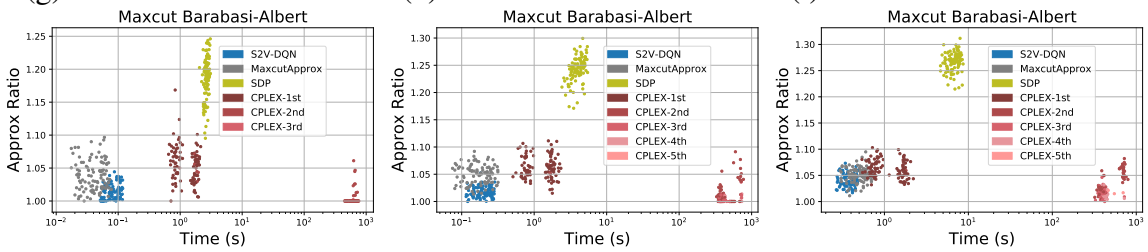
(a) MVC ER 50-100 (b) MVC ER 100-200 (c) MVC ER 200-300



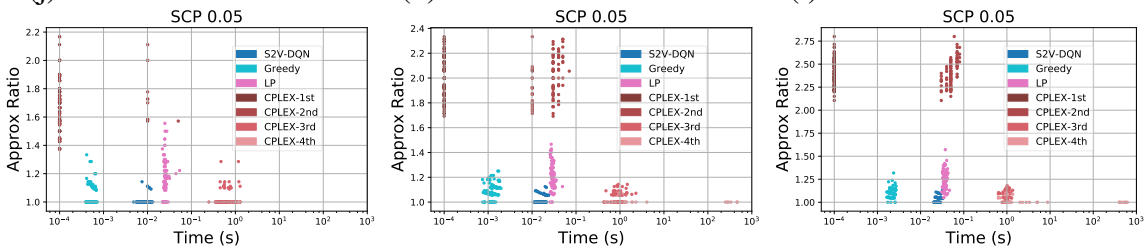
(d) MVC BA 50-100 (e) MVC BA 100-200 (f) MVC BA 200-300



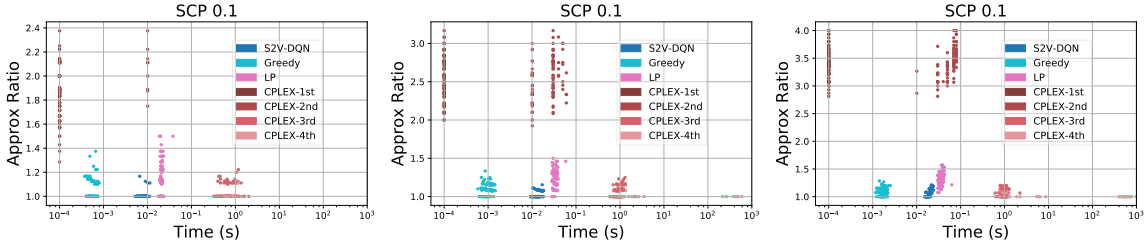
(g) MAXCUT ER 50-100 (h) MAXCUT ER 100-200 (i) MAXCUT ER 200-300



(j) MAXCUT BA 50-100 (k) MAXCUT BA 100-200 (l) MAXCUT BA 200-300



(m) SCP 0.05 50-100 (n) SCP 0.05 100-200 (o) SCP 0.05 200-300



(p) SCP 0.1 50-100 (q) SCP 0.1 100-200 (r) SCP 0.1 200-300

Figure A.3: Time-approximation trade-off for MVC, MAXCUT and SCP.

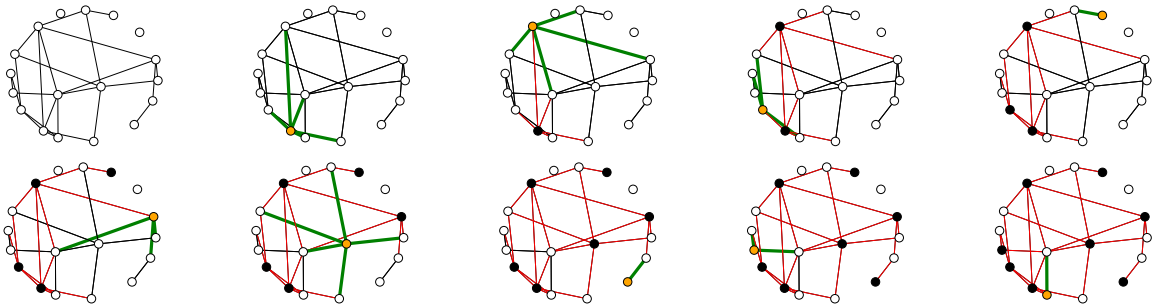


Figure A.4: Minimum Vertex Cover: an optimal solution to an ER graph instance found by S2V-DQN. Selected node in each step is colored in orange, and nodes in the partial solution up to that iteration are colored in black. Newly covered edges are in thick green, previously covered edges are in red, and uncovered edges in black. We show that the agent is not only picking the node with large degree, but also trying to maintain the connectivity after removal of the covered edges. For more detailed analysis, please see Appendix A.3.10.

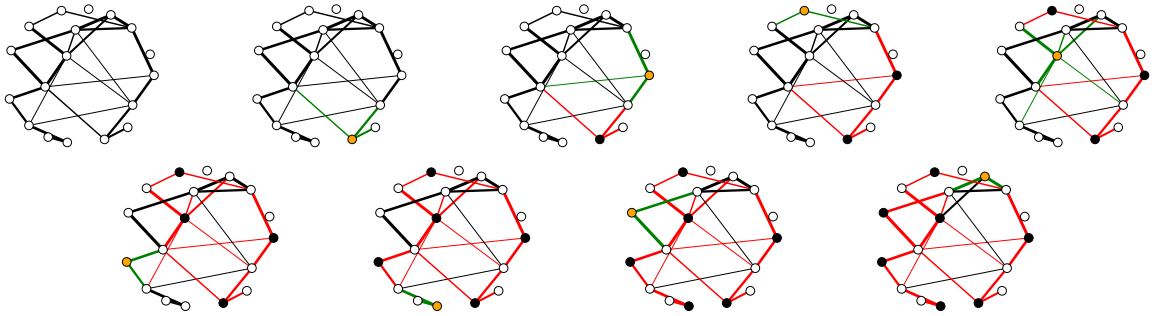


Figure A.5: Maximum Cut: an optimal solution to ER graph instance found by S2V-DQN. Nodes are partitioned into two sets: white or black nodes. At each iteration, the node selected to join the set of black nodes is highlighted in orange, and the new cut edges it produces are in green. Cut edges from previous iteration are in red (Best viewed in color). It seems the agent will try to involve the nodes that won't cancel out the edges in current cut set.

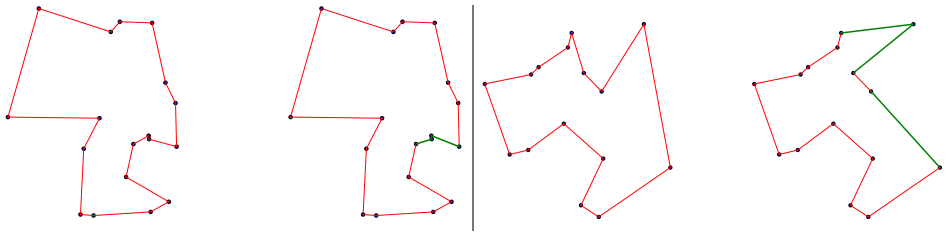


Figure A.6: Traveling Salesman Problem. Left: optimal tour to a "random" instance with 18 points (all edges are red), compared to a tour found by our method next to it. For our tour, edges that are not in the optimal tour are shown in green. Our tour is 0.07% longer than an optimal tour. Right: a "clustered" instance with 15 points; same color coding as left figure. Our tour is 0.5% longer than an optimal tour. (Best viewed in color).

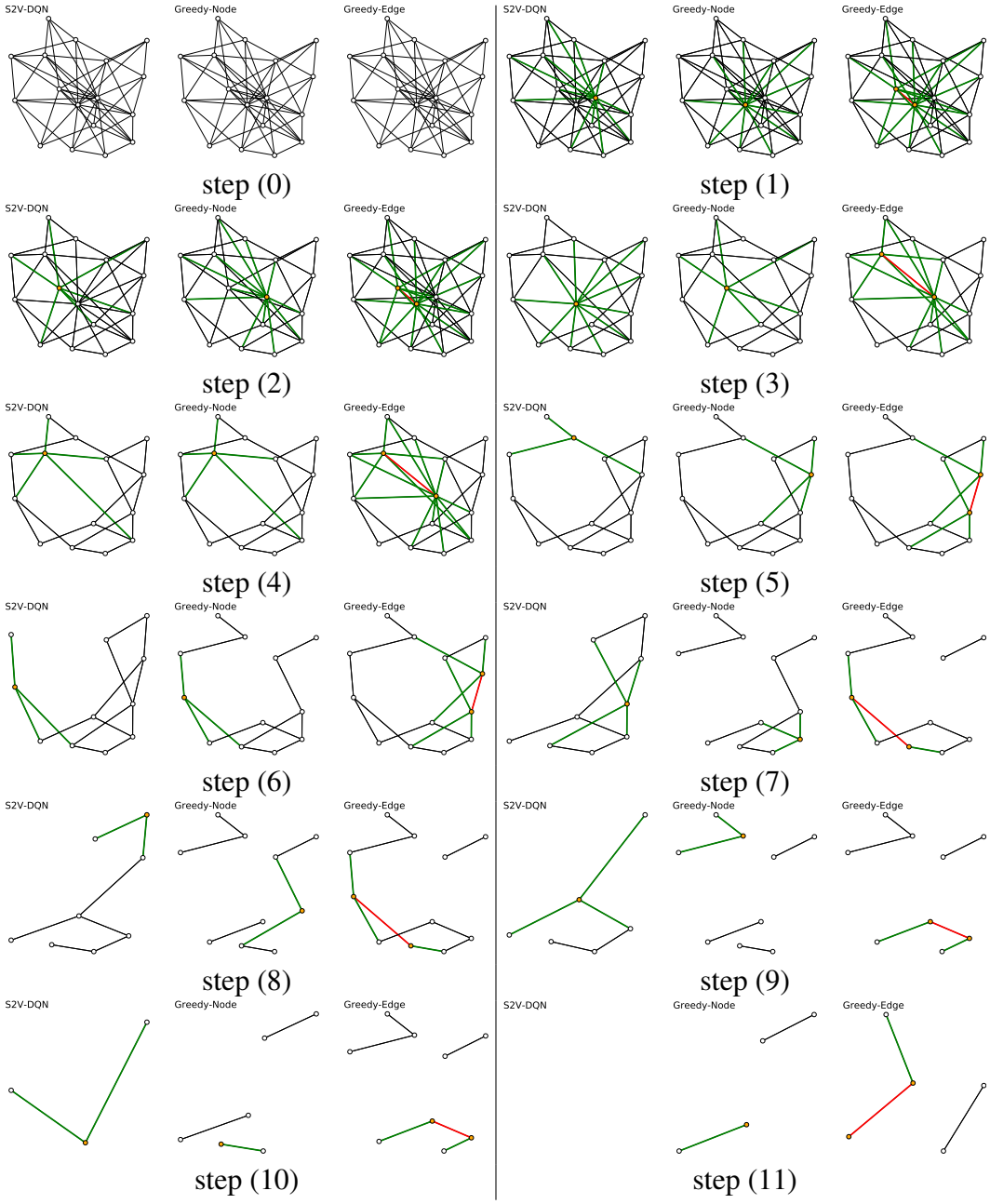


Figure A.7: Step-by-step comparison between our S2V-DQN and two greedy heuristics. We can see our algorithm will also favor the large degree nodes, but it will also do something smartly: instead of breaking the graph into several disjoint components, our algorithm will try the best to keep the graph connected.

REFERENCES

[1] D. J. Abraham, A. Blum, and T. Sandholm, "Clearing algorithms for barter exchange markets: Enabling nationwide kidney exchanges," in *Proceedings of the 8th ACM conference on Electronic commerce*, ACM, 2007, pp. 295–304.

[2] T. Achterberg, "Constraint integer programming," PhD thesis, Technische Universität Berlin, 2009.

[3] T. Achterberg and T. Berthold, "Improving the feasibility pump," *Discrete Optimization*, vol. 4, no. 1, pp. 77–86, 2007.

[4] ——, "Hybrid branching," in *CPAIOR*, Springer, 2009, pp. 309–311.

[5] T. Achterberg, T. Berthold, and G. Hendel, "Rounding and propagation heuristics for mixed integer programming," in *Operations Research Proceedings 2011*, Springer, 2012, pp. 71–76.

[6] T. Achterberg, T. Koch, and A. Martin, "Branching rules revisited," *Operations Research Letters*, vol. 33, no. 1, pp. 42–54, 2005.

[7] T. Achterberg and R. Wunderling, "Mixed integer programming: Analyzing 12 years of progress," in *Facets of Combinatorial Optimization*, M. Jünger and G. Reinelt, Eds., Springer, 2013.

[8] S. Ahmed, "Two-stage stochastic integer programming: A brief introduction," *Wiley Encyclopedia of Operations Research and Management Science*, 2010.

[9] S. Albers, "Online algorithms: A survey," *Mathematical Programming*, vol. 97, no. 1-2, pp. 3–26, 2003.

[10] R. Albert and A.-L. Barabási, "Statistical mechanics of complex networks," *Reviews of modern physics*, vol. 74, no. 1, p. 47, 2002.

[11] H. Alemdar, V. Leroy, A. Prost-Boucle, and F. Pétrot, "Ternary neural networks for resource-efficient ai applications," in *Neural Networks (IJCNN), 2017 International Joint Conference on*, IEEE, 2017, pp. 2547–2554.

[12] A. M. Alvarez, Q. Louveaux, and L. Wehenkel, "A machine learning-based approximation of strong branching," *INFORMS Journal on Computing*, vol. 29, no. 1, pp. 185–195, 2017.

[13] R. Anderson, J. Huchette, C. Tjandraatmadja, and J. P. Vielma, "Strong convex relaxations and mixed-integer programming formulations for trained neural networks," *arXiv preprint arXiv:1811.01988*, 2018.

[14] M. Andrychowicz, M. Denil, S. Gomez, M. W. Hoffman, D. Pfau, T. Schaul, and N. de Freitas, "Learning to learn by gradient descent by gradient descent," in *Advances in Neural Information Processing Systems*, 2016, pp. 3981–3989.

[15] D. Applegate, R. Bixby, V. Chvatal, and W. Cook, *Concorde TSP solver*, 2006.

[16] D. L. Applegate, R. E. Bixby, V. Chvatal, and W. J. Cook, *The traveling salesman problem: a computational study*. Princeton university press, 2011.

[17] E. Balas and A. Ho, "Set covering algorithms using cutting planes, heuristics, and subgradient optimization: A computational study," *Combinatorial Optimization*, pp. 37–60, 1980.

[18] C. Barnhart, N. L. Boland, L. W. Clarke, E. L. Johnson, G. L. Nemhauser, and R. G. Shenoi, "Flight string models for aircraft fleeting and routing," *Transportation science*, vol. 32, no. 3, pp. 208–220, 1998.

[19] I. Bello, H. Pham, Q. V. Le, M. Norouzi, and S. Bengio, "Neural combinatorial optimization with reinforcement learning," *arXiv preprint arXiv:1611.09940*, 2016.

[20] Y. Bengio, "How auto-encoders could provide credit assignment in deep networks via target propagation," *arXiv preprint arXiv:1407.7906*, 2014.

[21] Y. Bengio, A. Lodi, and A. Prouvost, "Machine learning for combinatorial optimization: A methodological tour d'horizon," *arXiv preprint arXiv:1811.06128*, 2018.

[22] T. Berthold, "Primal Heuristics for Mixed Integer Programs," Master's thesis, Technische Universität Berlin, 2006.

[23] T. Berthold, A. Lodi, and D. Salvagnin, "Ten years of feasibility pump, and counting," *EURO Journal on Computational Optimization*, pp. 1–14,

[24] D. Bertsimas, J. Pauphilet, and B. Van Parys, "Sparse classification and phase transitions: A discrete optimization perspective," *arXiv preprint arXiv:1710.01352*, 2017.

[25] D. Bertsimas and B. Van Parys, "Sparse high-dimensional regression: Exact scalable algorithms and phase transitions," *arXiv preprint arXiv:1709.10029*, 2017.

[26] B. Biggio, I. Corona, D. Maiorca, B. Nelson, N. Šrndić, P. Laskov, G. Giacinto, and F. Roli, "Evasion attacks against machine learning at test time," in *Joint European conference on machine learning and knowledge discovery in databases*, Springer, 2013, pp. 387–402.

[27] P. Bonami, A. Lodi, and G. Zarpellon, "Learning a classification of mixed-integer quadratic programming problems," in *International Conference on the Integration of Constraint Programming, Artificial Intelligence, and Operations Research*, Springer, 2018, pp. 595–604.

[28] J. Boyan and A. W. Moore, "Learning evaluation functions to improve optimization by local search," *Journal of Machine Learning Research*, vol. 1, no. Nov, pp. 77–112, 2000.

[29] T. C. Bressoud, R. Rastogi, and M. A. Smith, "Optimal configuration for bgp route selection," in *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies*, IEEE, vol. 2, 2003, pp. 916–926.

[30] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Security and Privacy (SP), 2017 IEEE Symposium on*, IEEE, 2017, pp. 39–57.

[31] M. Colombi, R. Mansini, and M. Savelsbergh, "The generalized independent set problem: Polyhedral analysis and solution approaches," *European Journal of Operational Research*, 2016.

[32] M. Courbariaux, I. Hubara, D. Soudry, R. El-Yaniv, and Y. Bengio, "Binarized neural networks: Training deep neural networks with weights and activations constrained to+ 1 or-1," *arXiv preprint arXiv:1602.02830*, 2016.

[33] H. Dai, B. Dai, and L. Song, "Discriminative embeddings of latent variable models for structured data," in *ICML*, 2016.

[34] H. Dai, E. B. Khalil, Y. Zhang, B. Dilkina, and L. Song, "Learning combinatorial optimization algorithms over graphs," in *Advances in Neural Information Processing Systems (NIPS)*, 2017.

[35] E Danna, "Performance variability in mixed integer programming," Presented at Workshop on Mixed Integer Programming, Columbia University, New York, 2008.

[36] M. Dawande, J. Kalagnanam, P. Keskinocak, F. S. Salman, and R Ravi, "Approximation algorithms for the multiple knapsack problem with assignment restrictions," *Journal of combinatorial optimization*, vol. 4, no. 2, pp. 171–186, 2000.

[37] S. S. Dey, A. Iroume, M. Molinaro, and D. Salvagnin, "Improving the randomization step in feasibility pump," *SIAM Journal on Optimization*, vol. 28, no. 1, pp. 355–378, 2018.

[38] B. Dilkina and C. P. Gomes, "Solving connected subgraph problems in wildlife conservation," in *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*, Springer, 2010, pp. 102–116.

[39] G. Dobson and R. S. Nambimadom, "The batch loading and scheduling problem," *Operations research*, vol. 49, no. 1, pp. 52–65, 2001.

[40] E. D. Dolan and J. J. Moré, "Benchmarking optimization software with performance profiles," *Mathematical programming*, vol. 91, no. 2, pp. 201–213, 2002.

[41] N. Du, L. Song, M. Gomez-Rodriguez, and H. Zha, "Scalable influence estimation in continuous-time diffusion networks," in *NIPS*, 2013.

[42] P. Erdos and A Rényi, "On the evolution of random graphs," *Publ. Math. Inst. Hungar. Acad. Sci*, vol. 5, pp. 17–61, 1960.

[43] M. Fischetti, F. Glover, and A. Lodi, "The feasibility pump," *Mathematical Programming*, vol. 104, no. 1, pp. 91–104, 2005.

[44] M. Fischetti and J. Jo, "Deep neural networks and mixed integer linear optimization," *Constraints*, vol. 23, pp. 296–309, 2018.

[45] M. Fischetti and M. Monaci, "Branching on nonchimerical fractionalities," *Operations Research Letters*, vol. 40, no. 3, pp. 159–164, 2012.

[46] L. Fleischer, M. X. Goemans, V. S. Mirrokni, and M. Sviridenko, "Tight approximation algorithms for maximum general assignment problems," in *Proceedings of the seventeenth annual ACM-SIAM symposium on Discrete algorithm*, Society for Industrial and Applied Mathematics, 2006, pp. 611–620.

[47] A. L. Friesen and P. Domingos, "Deep learning as a mixed convex-combinatorial optimization problem," *arXiv preprint arXiv:1710.11573*, 2017.

[48] A. Galloway, G. W. Taylor, and M. Moussa, "Attacking binarized neural networks," *arXiv preprint arXiv:1711.00449*, 2017.

[49] G. Gamrath, T. Fischer, T. Gally, A. M. Gleixner, G. Hendel, T. Koch, S. J. Maher, M. Miltenberger, B. Müller, M. E. Pfetsch, C. Puchert, D. Rehfeldt, S. Schenker, R. Schwarz, F. Serrano, Y. Shinano, S. Vigerske, D. Weninger, M. Winkler, J. T. Witt,

and J. Witzig, "The SCIP Optimization Suite 3.2," ZIB, Takustr.7, 14195 Berlin, Tech. Rep. 15-60, 2016.

[50] M. Goemans and D. P. Williamson, "Improved approximation algorithms for maximum cut and satisfiability problems using semidefinite programming," *Journal of the ACM*, vol. 42, no. 6, pp. 1115–1145, 1995.

[51] M. Gomez-Rodriguez, J. Leskovec, and A. Krause, "Inferring networks of diffusion and influence," in *Proceedings of the 16th ACM SIGKDD international conference on Knowledge discovery and data mining*, ACM, 2010, pp. 1019–1028.

[52] I. J. Goodfellow, J. Shlens, and C. Szegedy, "Explaining and harnessing adversarial examples," *arXiv preprint arXiv:1412.6572*, 2014.

[53] A. Graves, G. Wayne, M. Reynolds, T. Harley, I. Danihelka, A. Grabska-Barwińska, S. G. Colmenarejo, E. Grefenstette, T. Ramalho, J. Agapiou, *et al.*, "Hybrid computing using a neural network with dynamic external memory," *Nature*, vol. 538, no. 7626, pp. 471–476, 2016.

[54] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine, "Q-prop: Sample-efficient policy gradient with an off-policy critic," *arXiv preprint arXiv:1611.02247*, 2016.

[55] H. He and E. A. Garcia, "Learning from imbalanced data," *IEEE Transactions on knowledge and data engineering*, vol. 21, no. 9, pp. 1263–1284, 2009.

[56] H. He, H. Daume III, and J. M. Eisner, "Learning to search in branch and bound algorithms," in *Advances in Neural Information Processing Systems*, 2014, pp. 3293–3301.

[57] G. Hendel, *New rounding and propagation heuristics for mixed integer programming*, 2011.

[58] ——, "Empirical analysis of solving phases in mixed integer programming," Master's thesis, Technische Universität Berlin, 2014, p. 159.

[59] ——, "Enhancing mip branching decisions by using the sample variance of pseudo costs," in *Integration of AI and OR Techniques in Constraint Programming*, in press, vol. 9075, 2015, pp. 199 –214.

[60] D. S. Hochbaum and A. Pathria, "Forest harvesting and minimum cuts: A new approach to handling spatial constraints," *Forest Science*, vol. 43, no. 4, pp. 544–554, 1997.

[61] J. N. Hooker, "Testing heuristics: We have it all wrong," *Journal of Heuristics*, vol. 1, no. 1, pp. 33–42, 1995.

[62] F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Sequential model-based optimization for general algorithm configuration," in *International Conference on Learning and Intelligent Optimization*, Springer, 2011, pp. 507–523.

[63] F. Hutter, H. H. Hoos, K. Leyton-Brown, and T. Stützle, "ParamILS: An automatic algorithm configuration framework," *JAIR*, vol. 36, no. 1, pp. 267–306, 2009.

[64] F. Hutter, L. Xu, H. H. Hoos, and K. Leyton-Brown, "Algorithm runtime prediction: Methods & evaluation," *Artificial Intelligence*, vol. 206, pp. 79–111, 2014.

[65] IBM, *CPLEX User's Manual, Version 12.6.1*, version Version 12.6.1, 2014.

[66] S. Ioffe and C. Szegedy, "Batch normalization: Accelerating deep network training by reducing internal covariate shift," *arXiv preprint arXiv:1502.03167*, 2015.

[67] T. Joachims, "Optimizing search engines using clickthrough data," in *KDD*, 2002, pp. 133–142.

[68] ——, "Training linear SVMs in linear time," in *KDD*, 2006, pp. 217–226.

[69] D. S. Johnson and L. A. McGeoch, "Experimental analysis of heuristics for the stsp," in *The traveling salesman problem and its variations*, Springer, 2007, pp. 369–443.

[70] D. S. Johnson and M. A. Trick, *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*. American Mathematical Soc., 1996, vol. 26.

[71] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of computer computations*, Springer, 1972, pp. 85–103.

[72] F. K. Karzan, G. L. Nemhauser, and M. W. Savelsbergh, "Information-based branching schemes for binary linear mixed integer problems," *Mathematical Programming Computation*, vol. 1, no. 4, pp. 249–293, 2009.

[73] D. Kempe, J. Kleinberg, and É. Tardos, "Maximizing the spread of influence through a social network," in *KDD*, ACM, 2003, pp. 137–146.

[74] E. B. Khalil, B. Dilkina, and L. Song, "Scalable diffusion-aware optimization of network topology," in *Knowledge Discovery and Data Mining (KDD)*, 2014.

[75] E. B. Khalil, B. Dilkina, G. Nemhauser, S. Ahmed, and Y. Shao, "Learning to run heuristics in tree search," in *26th International Joint Conference on Artificial Intelligence (IJCAI)*, 2017.

[76] E. B. Khalil, A. Gupta, and B. Dilkina, "Combinatorial attacks on binarized neural networks," in *International Conference on Learning Representations (ICLR), arXiv:1810.03538 [cs.LG]*, 2019.

[77] E. B. Khalil, P. Le Bodic, L. Song, G. L. Nemhauser, and B. N. Dilkina, "Learning to branch in mixed integer programming.," in *AAAI*, 2016, pp. 724–731.

[78] E. B. Khalil, R. Trivedi, and B. Dilkina, "Neural integer optimization: Learning to satisfy generic constraints," in *submission to the International Conference on Machine Learning (ICML)*, 2019.

[79] D. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.

[80] J. Kleinberg and E. Tardos, *Algorithm design*. Pearson Education India, 2006.

[81] T. Koch, T. Achterberg, E. Andersen, O. Bastert, T. Berthold, R. E. Bixby, E. Danna, G. Gamrath, A. M. Gleixner, S. Heinz, *et al.*, "MIPLIB 2010," *Mathematical Programming Computation*, vol. 3, no. 2, pp. 103–163, 2011.

[82] J. Z. Kolter and E. Wong, "Provable defenses against adversarial examples via the convex outer adversarial polytope," *arXiv preprint arXiv:1711.00851*, 2017.

[83] M. Kruber, M. E. Lübbecke, and A. Parmentier, "Learning when to use a decomposition," in *International Conference on AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems*, Springer, 2017, pp. 202–210.

[84] A. Kurakin, I. Goodfellow, and S. Bengio, "Adversarial machine learning at scale," *arXiv preprint arXiv:1611.01236*, 2016.

[85] Y. Le Cun, "Learning process in an asymmetric threshold network," in *Disordered systems and biological organization*, Springer, 1986, pp. 233–240.

[86] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner, "Gradient-based learning applied to document recognition," *Proceedings of the IEEE*, vol. 86, no. 11, pp. 2278–2324, 1998.

[87] E. K. Lee, R. J. Gallagher, D. Silvern, C.-S. Wuu, and M. Zaider, "Treatment planning for brachytherapy: An integer programming model, two computational

approaches and experiments with permanent prostate implant planning," *Physics in Medicine and Biology*, vol. 44, no. 1, p. 145, 1999.

[88] J. T. Linderoth and M. W. Savelsbergh, "A computational study of search strategies for mixed integer programming," *INFORMS Journal on Computing*, vol. 11, no. 2, pp. 173–187, 1999.

[89] T.-Y. Liu, "Learning to rank for information retrieval," *Foundations and Trends in Information Retrieval*, vol. 3, no. 3, pp. 225–331, 2009.

[90] T.-Y. Liu, J. Xu, T. Qin, W. Xiong, and H. Li, "LETOR: Benchmark dataset for research on learning to rank for information retrieval," in *Proceedings of SIGIR Workshop on Learning to Rank for Information Retrieval*, 2007, pp. 3–10.

[91] A. Lodi and A. Tramontani, "Performance variability in mixed-integer programming," *Tutorials in Operations Research: Theory Driven by Influential Applications*, pp. 1–12, 2013.

[92] A. Madry, A. Makelov, L. Schmidt, D. Tsipras, and A. Vladu, "Towards deep learning models resistant to adversarial attacks," *arXiv preprint arXiv:1706.06083*, 2017.

[93] S. Martello, "Knapsack problems: Algorithms and computer implementations," *Wiley-Interscience series in discrete mathematics and optimiza tion*, 1990.

[94] MATLAB, *Statistics and Machine Learning Toolbox*. Natick, Massachusetts: The MathWorks Inc., 2013.

[95] B. McDanel, S. Teerapittayanon, and H. Kung, "Embedded binarized neural networks," in *Proceedings of the 2017 International Conference on Embedded Wireless Systems and Networks*, Junction Publishing, 2017, pp. 168–173.

[96] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[97] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[98] S. M. Moosavi Dezfooli, A. Fawzi, and P. Frossard, "Deepfool: A simple and accurate method to fool deep neural networks," in *Proceedings of 2016 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*, 2016.

[99] N. Narodytska, S. P. Kasiviswanathan, L. Ryzhyk, M. Sagiv, and T. Walsh, "Verifying properties of binarized deep neural networks," *arXiv preprint arXiv:1709.06662*, 2017.

[100] G. L. Nemhauser, "Integer programming: The global impact," 2013.

[101] G. L. Nemhauser and M. A. Trick, "Scheduling a major college basketball conference," *Operations Research*, vol. 46, no. 1, pp. 1–8, 1998.

[102] G. L. Nemhauser and L. A. Wolsey, *Integer and Combinatorial Optimization*. John Wiley & Sons, 1988.

[103] C. H. Papadimitriou and K. Steiglitz, *Combinatorial Optimization: Algorithms and Complexity*. New Jersey: Prentice-Hall, 1982.

[104] D. J. Papageorgiou, G. L. Nemhauser, J. Sokol, M.-S. Cheon, and A. B. Keha, "MIRPLib – A library of maritime inventory routing problem instances: Survey, core model, and benchmark results," *European Journal of Operational Research*, vol. 235, no. 2, pp. 350–366, 2014.

[105] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Security and Privacy (EuroS&P), 2016 IEEE European Symposium on*, IEEE, 2016, pp. 372–387.

[106] J. Patel and J. W. Chinneck, "Active-constraint variable ordering for faster feasibility of mixed integer linear programs," *Mathematical Programming*, vol. 110, no. 3, pp. 445–474, 2007.

[107] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and E. Duchesnay, "Scikit-learn: Machine learning in Python," *Journal of Machine Learning Research*, vol. 12, pp. 2825–2830, 2011.

[108] D. Peleg, G. Schechtman, and A. Wool, "Approximating bounded 0-1 integer linear programs," in *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*, IEEE, 1993, pp. 69–77.

[109] M. Rastegari, V. Ordonez, J. Redmon, and A. Farhadi, "Xnor-net: Imagenet classification using binary convolutional neural networks," in *European Conference on Computer Vision*, Springer, 2016, pp. 525–542.

[110] G. Reinelt, "Tsplib—a traveling salesman problem library," *ORSA journal on computing*, vol. 3, no. 4, pp. 376–384, 1991.

[111] M. Riedmiller, "Neural fitted q iteration–first experiences with a data efficient neural reinforcement learning method," in *European Conference on Machine Learning*, Springer, 2005, pp. 317–328.

[112] C. Rudin, "The p-norm push: A simple convex ranking algorithm that concentrates at the top of the list," *JMLR*, vol. 10, pp. 2233–2271, 2009.

[113] A. Sabharwal, H. Samulowitz, and C. Reddy, "Guiding combinatorial optimization with uct," in *Integration of AI and OR Techniques in Contraint Programming for Combinatorial Optimzation Problems*, Springer, 2012, pp. 356–361.

[114] H. Samulowitz and R. Memisevic, "Learning to solve QBF," in *AAAI*, 2007.

[115] T. Sandholm, "Very-large-scale generalized combinatorial multi-attribute auctions: Lessons from conducting $60 billion of sourcing," in *Handbook of Market Design*, N. Vulkan, A. E. Roth, and Z. Neeman, Eds., Oxford University Press, 2013, ch. 16.

[116] D. Selsam, M. Lamm, B. Bunz, P. Liang, L. d. Moura, and D. L. Dill, "Learning a sat solver from single-bit supervision," in *International Conference on Learning Representations*, 2019.

[117] A. Sinha, H. Namkoong, and J. Duchi, "Certifiable distributional robustness with principled adversarial training," *arXiv preprint arXiv:1710.10571*, 2017.

[118] K. A. Smith, "Neural networks for combinatorial optimization: A review of more than a decade of research," *INFORMS Journal on Computing*, vol. 11, no. 1, pp. 15–34, 1999.

[119] C. Spearman, "The proof and measurement of association between two things," *The American journal of psychology*, vol. 15, no. 1, pp. 72–101, 1904.

[120] R. Sutton and A. Barto, *Reinforcement Learning: An Introduction*. MIT Press, 1998.

[121] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I. Goodfellow, and R. Fergus, "Intriguing properties of neural networks," *arXiv preprint arXiv:1312.6199*, 2013.

[122] V. Tjeng, K. Xiao, and R. Tedrake, "Evaluating robustness of neural networks with mixed integer programming," *arXiv preprint arXiv:1711.07356*, 2017.

[123] J. Uesato, B. O'Donoghue, A. v. d. Oord, and P. Kohli, "Adversarial risk and the dangers of evaluating against weak attacks," *arXiv preprint arXiv:1802.05666*, 2018.

[124] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "Finn: A framework for fast, scalable binarized neural network in-

ference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, ACM, 2017, pp. 65–74.

[125] O. Vinyals, M. Fortunato, and N. Jaitly, "Pointer networks," in *Advances in Neural Information Processing Systems*, 2015, pp. 2692–2700.

[126] H. Xiao, K. Rasul, and R. Vollgraf. (Aug. 28, 2017). Fashion-mnist: A novel image dataset for benchmarking machine learning algorithms. arXiv: cs.LG/1708.07747 [cs.LG].

[127] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "Hydra-MIP: Automated algorithm configuration and selection for mixed integer programming," in *Proceedings of the 18th RCRA Workshop on Experimental Evaluation of Algorithms for Solving Problems with Combinatorial Explosion*, 2011, pp. 16–30.

[128] W. Zhang and T. G. Dietterich, "A reinforcement learning approach to job-shop scheduling," in *IJCAI*, Citeseer, vol. 95, 1995, pp. 1114–1120.

VITA

Elias Khalil is a Ph.D. candidate at the College of Computing at Georgia Tech. His research interests are in Artificial Intelligence with a focus on machine learning and discrete optimization. He is the recipient of an IBM Ph.D. Fellowship (2016-2017), the First Prize in the poster competition at INFORMS (2017) and the Best Paper Award at the NIPS Workshop on Frontiers of Network Analysis (2013). He has interned at IBM Research and Symantec Research Labs, and has received his MS from Georgia Tech (2014) and his BS from the American University of Beirut (2012), both in Computer Science. Elias was born and raised in the city of Beirut, capital of Lebanon.